NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

ROBOTIC MANIPULATION ON A MOVING PLATFORM UTILIZING FORCE SENSING AND SONAR RANGING

by

Roy A. Raphael

March 1998

Thesis Advisor:	Xiaoping Yun
Second Reader:	John G. Ciezki

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 1998	3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE ROBOTIC MANIPULATION ON A MOVING PLATFORM UTILIZING FORCE SENSING AND SONAR RANGING		5. FUNDING NUMBERS
6. AUTHOR(S) Raphael, Roy, A.		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Robotic manipulators are widely used in industry where the environment may be too hostile for workers. However, their application has been limited to an industrial setting where the robot is mounted on a stationary base. It is of great interest to expand the application of the robot manipulator to where it is mounted on an autonomous delivery vehicle. This application would enable the delivery vehicle not only to locate objects in a hostile environment, but also to perform tasks that would entirely remove the human being from the hostile environment. This thesis explores the feasibility of implementing a manipulator on an autonomous vehicle. A Zebra-ZERO Force Control Robot is mounted on a moving platform for feasibility simulations of an autonomous delivery vehicle. The Zebra-ZERO system consists primarily of a robotic arm with six degrees of freedom, a six-axis force sensor mounted at the end of the manipulator, and supporting computer hardware and software. In this thesis, the capability of the Zebra-ZERO system is expanded by integrating it with an external sonar ranging system. The sonar ranging system provides range feedback that is critical for positioning the manipulator while it is mounted on a moving platform. Test results demonstrate that the manipulator mounted on a moving platform is able to compensate for random platform motions and successfully perform various manipulation tasks.

14. SUBJECT TERMS Control, Zebra-ZERO, force sensor, sonar ranging, robot manipulator			15. NUMBER OF PAGES 132
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited

ROBOTIC MANIPULATION ON A MOVING PLATFORM UTILIZING FORCE SENSING AND SONAR RANGING

Roy A. Raphael
Lieutenant, United States Navy
B.S., University of San Diego, 1991

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 1998

Author: [illegible]
Roy A. Raphael

Approved by: [illegible]
Xiaoping Yun, Thesis Advisor

[illegible]
John G. Ciezki, Second Reader

[illegible]
Herschel H. Loomis, Jr., Chairman
Department of Electrical and Computer Engineering

ABSTRACT

Robotic manipulators are widely used in industry where the environment may be too hostile for workers. However, their application has been limited to an industrial setting where the robot is mounted on a stationary base. It is of great interest to expand the application of the robot manipulator to where it is mounted on an autonomous delivery vehicle. This application would enable the delivery vehicle not only to locate objects in a hostile environment, but also to perform tasks that would entirely remove the human being from the hostile environment. This thesis explores the feasibility of implementing a manipulator on an autonomous vehicle. A Zebra-ZERO Force Control Robot is mounted on a moving platform for feasibility simulations of an autonomous delivery vehicle. The Zebra-ZERO system consists primarily of a robotic arm with six degrees of freedom, a six-axis force sensor mounted at the end of the manipulator, and supporting computer hardware and software. In this thesis, the capability of the Zebra-ZERO system is expanded by integrating it with an external sonar ranging system. The sonar ranging system provides range feedback that is critical for positioning the manipulator while it is mounted on a moving platform. Test results demonstrate that the manipulator mounted on a moving platform is able to compensate for random platform motions and successfully perform various manipulation tasks.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

A. PROBLEM STATEMENT

Remote manipulation of objects with little or no human intervention is an important issue in modern robotics. It is especially critical in the areas of mine clearance, space exploration and construction and operation of the multinational space station [Ref. 21]. The Zebra-ZERO Force Control Robot [Ref. 1] supports the study of manipulating an object under the condition that the control system has prior knowledge of the approximate location of the object. The end-effector is first directed to a location that is in close proximity with the object. It then executes a hunting algorithm to put the end-effector in contact with the object. As contact is made, the force sensor is utilized to perform the desired task(s). The Zebra-ZERO system is not able to utilize a closed-loop control algorithm until the end-effector is in contact with the object. This poses an obvious constraint when operating on a moving platform.

Given that the Zebra-ZERO is mounted on a mobile platform, it is desirable to investigate the application and limitations of the manipulator. To aid in this study, the Zebra-ZERO must be equipped with a medium-range (1 to 4 feet) sensor system that provides positioning data for the manipulator control system while it performs tasks on a moving platform. Furthermore, the medium-range sensor system must provide real-time control information to the Zebra-ZERO system in a data format that is easily accessed by the control algorithm. In order to utilize an additional sensor, it is necessary to implement the following steps:

- Design and build an intermediate-range sensor system.
- Interface a medium-range sensor system to the Zebra-ZERO system.

- Develop algorithms that integrate the medium-range sensor into the Zebra-ZERO control software.

There are many types of medium-range sensors that may be used to facilitate this study, including lasers, and IR/Radio beacons. However, the sensor of choice in this study is the sonar transducer. A sonar transducer system is chosen because relatively fewer parts are required to provide real-time data to the Zebra-ZERO system resulting in lower cost. Furthermore, since the sonar sensor has the slowest range acquisition rate (10 updates per second) [Ref. 4] of the above mentioned sensors, use of faster sensors such as the Sensus 500 laser system (30 updates per second) [Ref. 13] can only enhance the performance of the Zebra-ZERO. Therefore, the test results utilizing a sonar transducer are the most portable to the other sensor systems.

Successful employment of the manipulator on a moving platform requires that sonar information is continually available to the Zebra-ZERO control software for real-time implementation. Therefore, the CyberResearch CYDIO 24 digital I/O board is used to input sonar ranging data. Installed in one of the Zebra-ZERO motherboard slots, the sonar ranging system sends continually updated ranging information to the I/O board where it is made available to the control software. Once the ranging information is retrieved, it is integrated into the control algorithm. Efficient and accurate control of the Zebra-ZERO on a moving platform is achieved through task-specific code developed to handle integrated control of both the medium-range sonar system and the force sensing system.

B. THESIS ORGANIZATION

This thesis reports on the feasibility of implementing the Zebra-ZERO force control robot while it is mounted on an autonomous vehicle. The Zebra-ZERO is a relatively inexpensive robot platform with a vast software library of control functions. This makes it an ideal choice for studying robotic manipulation while the robot is on a static base. However, it is of great interest to study how well the Zebra-ZERO system operates in an environment where the manipulator is mounted on a moving platform. To conduct this study, a sonar ranging system is integrated into the Zebra-ZERO system to provide range information that positions the manipulator in a dynamic operating environment.

Chapters II through VII document the development of variations of pick-and-place algorithms that utilize the combination of the Zebra-ZERO Robot and a sonar ranging system on a moving platform. Technical aspects of each component system are described in Chapter II to provide the reader with an understanding of the Zebra-ZERO, CYDIO 24 Digital I/O board, and Ultrasonic Ranging System. Applicable concepts of controller design are presented in Chapter III. In Chapter IV, the reader is introduced to Zebra-ZERO programming concepts that include writing, compiling, linking, and running programs from within a Borland Turbo **C++** 3.0 programming shell. Also, a segment of the chapter introduces basic and more advanced programs which are explained in detail to clarify the applications of the Zebra-ZERO software library. Chapter V contains a description of the hardware simulations that employ the Zebra-ZERO on a moving platform and documents the results of the simulations. Concluding remarks and proposed recommendations for future work are presented in Chapter VI.

II. BACKGROUND AND SYSTEM DESCRIPTION

A. THE ZEBRA-ZERO SYSTEM

The Zebra-ZERO Force Control Robot, built by Integrated Motions, Incorporated, is designed for robotics research and development. Recent research projects include the development of fine-motion planning systems that utilize the force sensing [Ref. 14], the development of general motion planning systems for assembly tasks [Ref. 15], and contact space analysis [Ref. 16].

The robot arm is composed of six revolute joints that allow the manipulator to position objects/tools with arbitrary orientations within a characteristic workspace. The last link of the manipulator is attached to a six-axis force sensor which in turn is attached to the gripper. The Zebra-ZERO system is controlled from a DOS-based personal computer (PC) utilizing C programs and libraries.

The entire Zebra-ZERO system consists of the manipulator, gripper, force sensor, power amplifiers/power supply, motor control board, PC, and robot control software. The individual elements work together to produce coordinated user defined motions and positioning solutions.

1. PC and Robot Control Software

The Zebra-ZERO utilizes a PC as the user-hardware interface. It contains a 133 MHz Pentium processor and implements the DOS operating system. Control, monitoring, and code execution are implemented through C coded software that includes a library of functions provided by Integrated Motions, Inc. [Ref. 1]

The system delivered with a Borland Turbo **C++** compiler and a compiled library of robot control functions called *robot.lib.* The library contains functions that execute a wide variety of tasks such as kinematic routines, path planners, servo code, motion parameter setting, and high-level motion/manipulation commands. The programmer is able to utilize **C** code and the vast library of functions to create a broad spectrum of task-specific programs and functions [Ref. 1]. For instance, the programmer can design code that controls the manipulator so that it approaches a surface while it polls the force sensor output to detect contact. After the end-effector contacts the surface, it can perform programmer-defined tasks such as removing a peg or moving along a surface. Examples of these type of programs are presented in chapters IV and V.

2. Arm Assembly

The Zebra-ZERO manipulator is a six-link, six-joint, revolute, rigid mechanical manipulator with six degrees of freedom. Its purpose is to impose the system control law on the environment. Additionally, the manipulator houses all six drive motors with their associated linkages and optical position encoders. The force sensor with attached gripper is mounted at the end of the wrist joint. Figure 1 is a kinematic representation of the Zebra-ZERO manipulator. The links and joints are referred to as **L1** through **L6**, and **J1** through **J6** respectively. The fixed base plate is assumed to be **L0** and is not shown in Figure 1. **L0** is connected to **L1** which is the rotating base carriage. **L1** through **L6** are connected in order through their respective joints, **J1** through **J6** [Ref. 1].

The plane in which the base carriage (**J1** axis) lies and to which the upper arm (**J2** axis) is normal is defined as the *plane of the arm*. This plane contains both the upper arm and forearm links. The **J3** axis intersects and is orthogonal to the **J4** axis. The **J4**, **J5**, and **J6** axes all intersect at the same point. This intersection point is defined as the *wrist center* [Ref. 1].

The three primary Cartesian frames are also shown in Figure 1. They are the *base frame*, *wrist frame*, and the *tool frame*. The base frame is fixed and is imbedded in the intersection of the **J1** and **J2** axes such that the **x**-axis points forward and away from the manipulator cables and is located eleven inches above the mounting surface. The origin of the wrist frame is in the intersection of the wrist axes and is rigidly attached to the force sensor and gripper. The tool frame, in this case, is depicted as being located at the fingertips of the gripper which is the default location [Ref. 1].

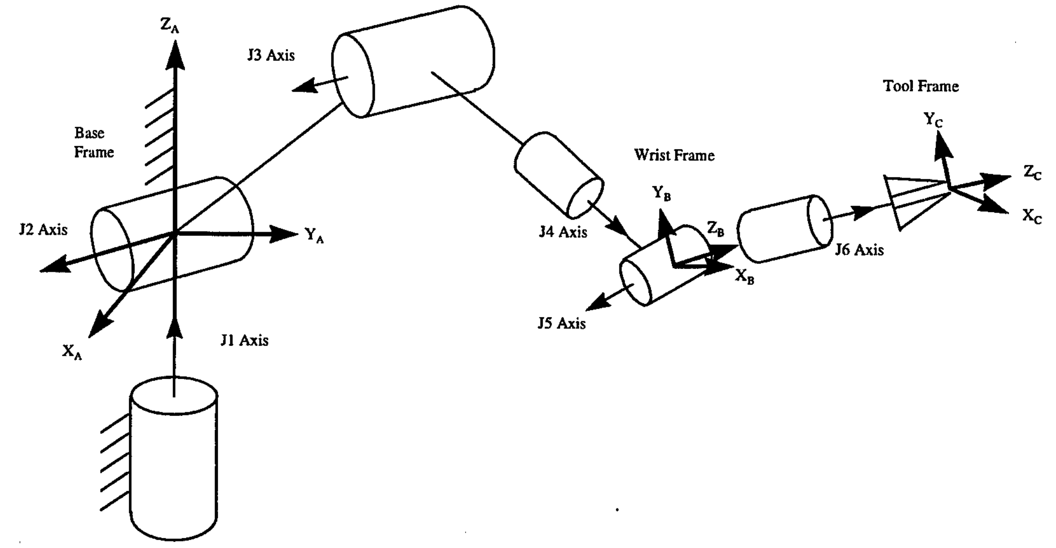

Figure 1. Kinematic Configuration. Ref. [1]

3. Drive System

The Zebra-ZERO drive system consists of six DC brush-commutated motors and their associated combinations of shaft and gear linkages. The motors are energized by the Power Amplifier Board described in a subsequent section. Each motor drives an individual joint through the drive linkages that run throughout the manipulator's hollow links. Each motor has a two-stage planetary gear head with gear reductions of 24:1, and an optical encoder. Motors that drive **J1** and **J2** are mounted in the rotating base carriage and transmit power directly through a set of gears to their respective joints. The motors that drive **J3** through **J6** are mounted in the upper arm. Power for the wrist joints is transmitted via shafts that run through the arm and bevel gears located at the elbow joint. The wrist uses a concentric shaft differential to drive its three intersection joints. The optical encoders are mounted directly to each motor shaft and provide position feedback to the motor controller board [Ref. 1].

4. Motor Control Board

The motor control board is the arbitrator between the PC and the manipulator. Its primary task is to execute motor control by measuring shaft angles from each motor encoder and outputting motor commands to the power amplifiers [Ref. 1].

The HCTL-1 motor control board was designed and built by Hewlett Packard for general purpose motion control. It utilizes eight HCTL-1100 general purpose motion control integrated circuits (ICs). The purpose of the ICs is to free the PC for other tasks by performing the time-intensive functions of digital motion control [Ref. 2]. Seven of the eight channels are used by the Zebra-ZERO with the eighth channel left as a spare. Six HCTL-1100's are used to control each of the six degrees of freedom, and one is used to control the gripper [Ref. 1].

The motor control board interfaces with the computer through a full-length slot in the PC's motherboard. Since the HCTL-1 is a memory mapped device, the registers for each chip are mapped directly into the PC's memory starting at the HCTL-1 segment address. Control modes are programmed by writing directly to the respective 64-bit register. The segment address is set on the board itself through eight dip switches which correspond to address bits A10 through A17. Address bits A18 and A19 are hardwired to logical 1. The segment address of the board in the Zebra-ZERO is set to 0xD000 [Ref. 1].

The memory address for successive HCTL-1100s are located sequentially in memory and are referenced as axes 0 through 7 [Ref. 1]. In order to write to a particular HCTL 1100 address, the memory call must be made to the sum of the segment address, chip offset, and chip register offset. A typical function call in **C** code that writes a logical 1 to the tenth register of controller axis 2 with the segment address set to 0xD000 is as follows:

pokeb(0xD000,0x8A,1).

The HCTL-1 motor controller outputs seven power amplifier control channels and inputs seven position encoder feedback signals. Each output is extracted from a single HCTL-1100 IC and consists of a pulse-width modulated (PWM) speed and direction signal. The PWM speed signal is a 20 KHz square wave whose duty cycle controls how much current is to be applied to the respective motor. A duty cycle of 0% commands maximum current and a duty cycle of 100% commands minimum current. Each IC also has a TTL level optical encoder input that provides angular joint position feedback to the control algorithm. The physical connection between the motor controller board and the power amplifier board consists of two 40-pin and one 10-pin flat ribbon cables. These cables provide the speed and direction commands for each motor, speed and direction commands for the gripper, the position feedback from each position encoder, and the output data from the force sensor. [Ref. 1]

5. Power Amplifiers and Power Supply

The Zebra-ZERO power supply provides power to all of the joint motors through the power amplifier board. The power supply is located in its own enclosure and provides 24 volt DC, 23 amp unregulated power. It has a power-off, and a power-on switch mounted on its enclosure as well as a jack for a remote power off switch. Power on is indicated by illumination of a green power on light. The power supply output goes directly to the base of the arm assembly where it connects to the power amplifier board [Ref. 1].

The power amplifier board has a power amplifier for each joint motor as well as the gripper and one extra unused amplifier. The board is housed in the stationary portion of the arm base assembly. Each PWM power amplifier receives 24 volts at 3 amps which is provided by the power supply. Their respective power transistors are attached to a heat sink that is bolted to the top of the power amplifier housing. The power amplifier logic is designed to drive the joint motors in accordance with the PWM input signals. The amplifiers shut down when a control signal is absent. The power amplifier board also accepts encoder and force sensor signals that it passes on to the motor controller board [Ref. 1].

6. Gripper

The Zebra-ZERO utilizes an electric gripper as an end-effector. The gripper allows the manipulator to grasp and manipulate objects within the working environment. The gripper actuator consists of a screw-type mechanism that opens and closes the fingers within a range of 0.0 to 85.0 millimeters [Ref. 1].

7. Force Sensor

The force sensor is a cylindrical device mounted between the wrist flange and the gripper. The sensor is instrumented with strain gauges that measure the forces and torques acting on the end-effector. The Zebra-ZERO force sensor provides six strain measurements. They represent forces and moments acting in the directions shown in Figure 2.

The data acquisition system within the sensor receives strain signals from three pairs of strain gauges each mounted on three individual bending beams within the sensor. The beams flex as forces and moments are applied to the end-effector. Strain measurements are extracted and used to construct a 6x1 strain measurement vector whose element values are proportional to those of the applied forces and moments [Ref. 1]. The maximum force reading is 15 grams. However, forces over 100 Kgf, or moments grater than 4000 Kgf-mm will damage the sensor [Ref. 1].

The six strain measurement signals are fed into a data acquisition system within the sensor where they are digitized and sent serially to the power amplifier board. From the power amplifier board, the signal is sent to the motor controller card were the data is reassembled into parallel words and made available to the PC as a 6x1 force/moment vector in the tool frame [Ref. 1].

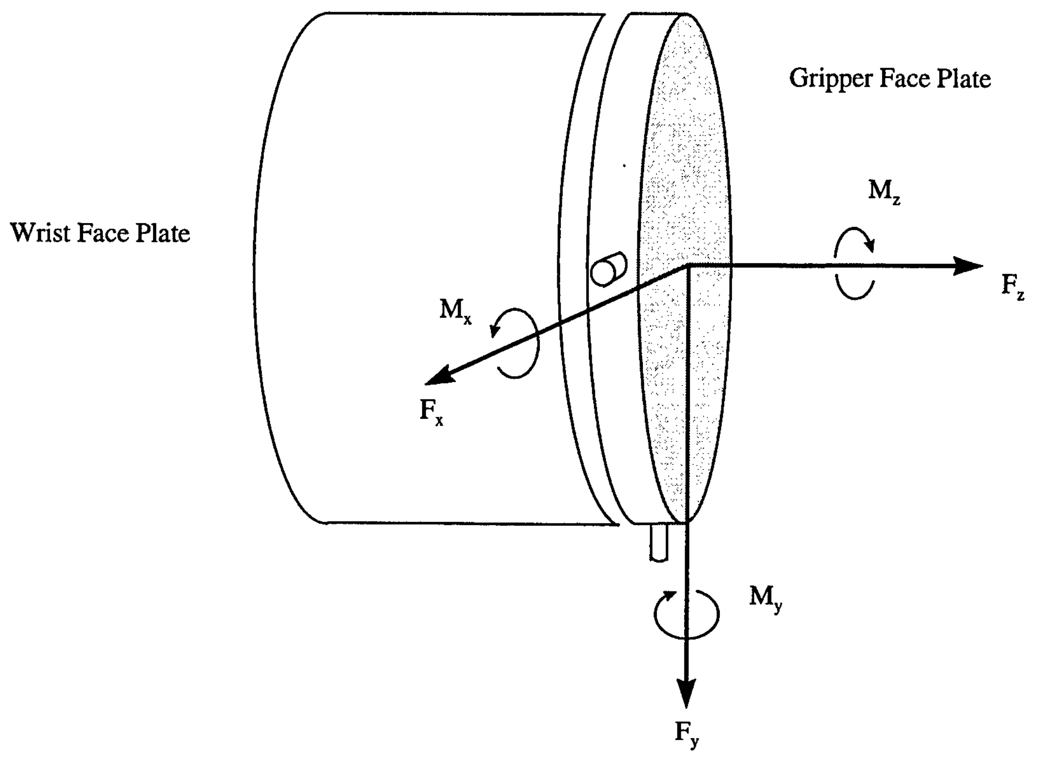

Figure 2. Force Sensor

Power for the force sensor is provided by the PC power supply. This feature allows the user to access force sensor information even when power is not applied to the arm assembly.

B. ULTRASONIC RANGING SYSTEM

The Ultrasonic Ranging System provides the Zebra-ZERO control algorithm with range data from entities in the control environment. Sonar ranging data is the control input that safely guides the arm assembly toward the target object. Accurate and frequent range feedback facilitates timely control of the manipulator so as to reduce the possibility of damaging the manipulator assembly or the prime mover while minimizing the closing time to the target object.

Figure 3 is the block diagram of the Ultrasonic Ranging System. It consists of the Polaroid Ranging Unit, Ultrasonic Transducer, and Ranging Controller Circuit (RCC). The ranging system works autonomously to acquire sonar ranging data and to continuously pass the data to the Digital I/O board.

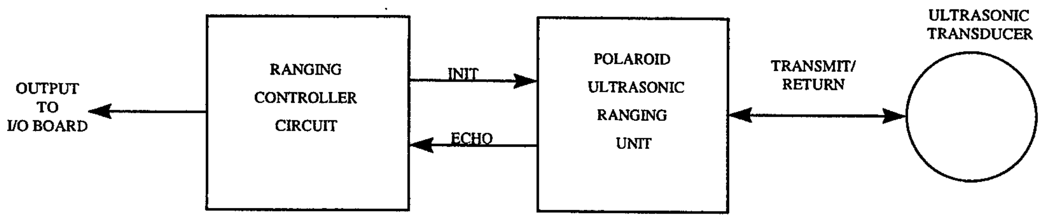

Figure 3. Ultrasonic Ranging System Block Diagram

1. Polaroid Ultrasonic Ranging Unit

The Polaroid Ultrasonic Ranging Unit is designed and built by the Polaroid Corporation for the sole purpose of controlling the operating mode (transmit/receive) of the sonar transducer [Ref. 4]. The ranging unit consists of the power interface, digital, analog, and coupling circuits. They all work together to implement the sonar transducer as shown in Figure 4.

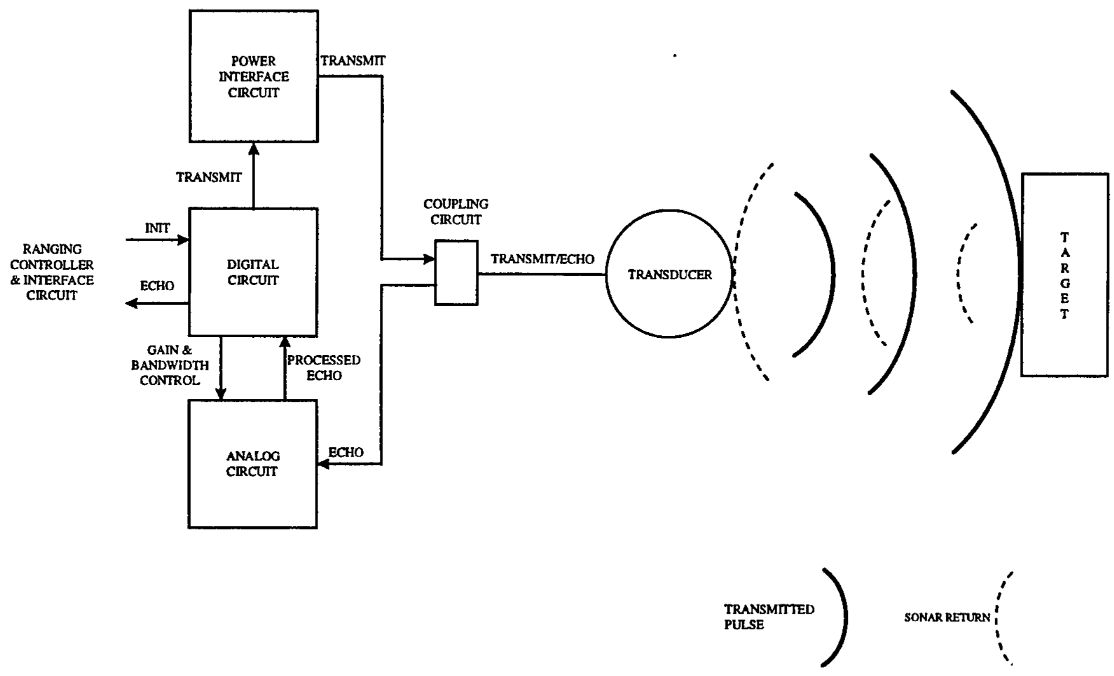

Figure 4. Ranging Circuit Board Block Diagram. Ref. [4]

The digital section of the ranging circuit board receives the Input Transmit Command (INIT) from the RCC. Upon receipt of INIT, the digital circuit creates a low-power modulated electrical pulse that is sent to the power interface circuit. The power interface circuit then generates a high-energy electrical pulse which is sent to the transducer. While transmitting, the transducer appears to be a loudspeaker to the Ranging Unit. The transducer converts the high energy electrical signal into an ultrasonic "Chirp." After the pulse is sent and a short blanking period is invoked, the analog circuit is enabled so that it can receive the sonar return from the target. The transducer now appears to be a microphone to the Ranging Unit. Once the return is received by the transducer and processed by the analog circuit, the raw unprocessed Echo (ECHO) signal is sent to the digital section. The digital section then converts ECHO into a square wave which is sent to the ranging controller circuit. [Ref. 4]

The circuit board utilizes a nine-pin plug with which to interface to the control circuit. However, only seven of the nine pins are utilized. The signals available at the pins are described in Table 1.

Pin Number	Signal	Description
1	GND	Ground (GND): Module ground line.
2	BLINK	Blanking (BLINK): Utilized in multiple-echo mode. After the first echo is received and ECHO is set high, BLINK must be taken high then low to reset the ECHO output for the next echo to be detected. BLINK is set low for this application.
3	Not Used	N/A
4	INIT	Transmit Initiate (INIT): Transition from low to high triggers the Transmit pulse.
5	Not Used	N/A
6	OSC	Oscillator (OSC): The oscillator onboard the module generates a 420 KHz signal as a time base for the modulated pulse. OSC is an output based on the oscillator output and is provided for external use. OSC is not used in this application.
7	ECHO	Echo (ECHO): transitioning from high to low indicates the time a reflected signal is received by the transducer. The time between INIT going high and the ECHO output going high is proportional to the distance between the target and the transducer.
8	BINH	Blanking Inhibit (BINH): BINH High ends the blanking of the receive input prior to internal blanking. BINH is set low for the mode of operation (Single Echo) used in this application.
9	VCC	VCC: 6 VDC, 2.5 amp power supply.

Table 1. 6500 Series Sonar Ranging Module Inputs and Outputs

2. Ranging Controller Circuit

The Ranging Controller Circuit (RCC) generates INIT, processes ECHO, provides power to the Polaroid Ultrasonic Ranging Unit, and outputs an eight-bit digital range measurement. The RCC requires a 6 VDC, 2.5 amp power supply to drive both the controller circuitry and the Ranging Unit. INIT is sent to the Ranging Unit to trigger the sonar transmission signal. The Ranging Unit sends ECHO to the RCC to signal the end of the ranging cycle. The RCC computes the time difference between the generation of INIT and the reception of ECHO, from which it calculates an eight-bit word containing the measured time it takes the sonar transmission to travel from the transducer to the target and back to the transducer. The eight-bit binary range output is displayed by a bank of LEDs on the RCC and is sent to the I/O board for processing by the PC.

The RCC design has six major sections. The sections work together to generate control signals which support both operating the Polaroid Ultrasonic Ranging Unit in single-echo mode and outputting accurate ranging data to the PC. Relevant technical information is presented in Figures 5 through 7. Figure 5 is the RCC block diagram, Figure 6 is the RCC timing diagram, and Figure 7 is the RCC schematic diagram.

The drive circuit generates INIT, a 5 Hz square wave that signals the beginning of each ranging cycle and triggers the transmission of each sonar pulse. INIT is sent to the Ranging Unit to initiate a sonar transmission and to the reference latch to clear the binary counter. After applying power (V_{cc}) to the Ranging Unit, a minimum of 5 milliseconds must elapse before the Ultrasonic Ranging Unit receives INIT [Ref. 4, p.19]. The RCC is equipped with a DIP switch that the operator closes after power is applied that allows INIT to be applied to the Ranging Unit.

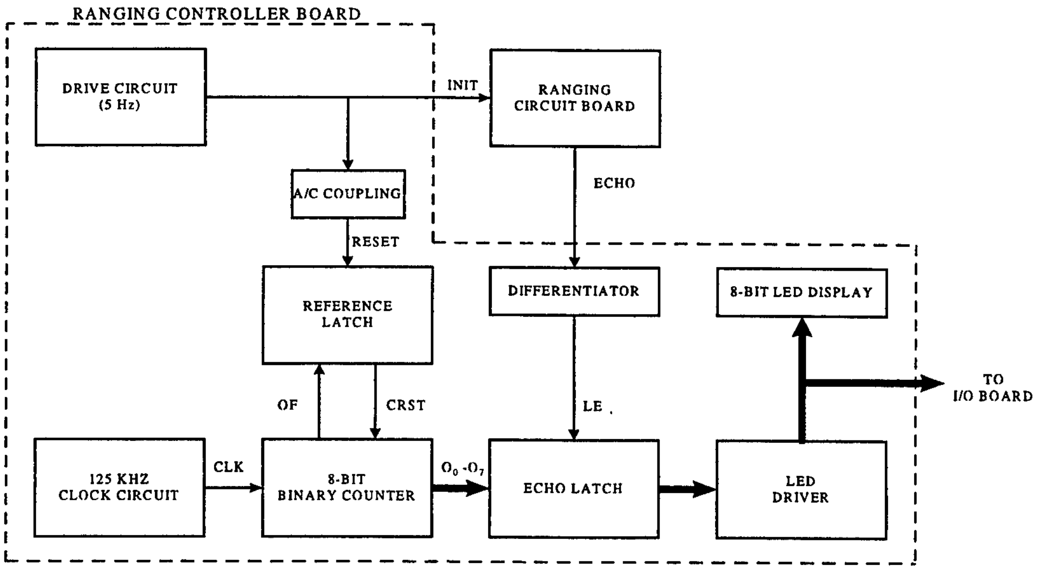

Figure 5. Ranging Controller Block Diagram

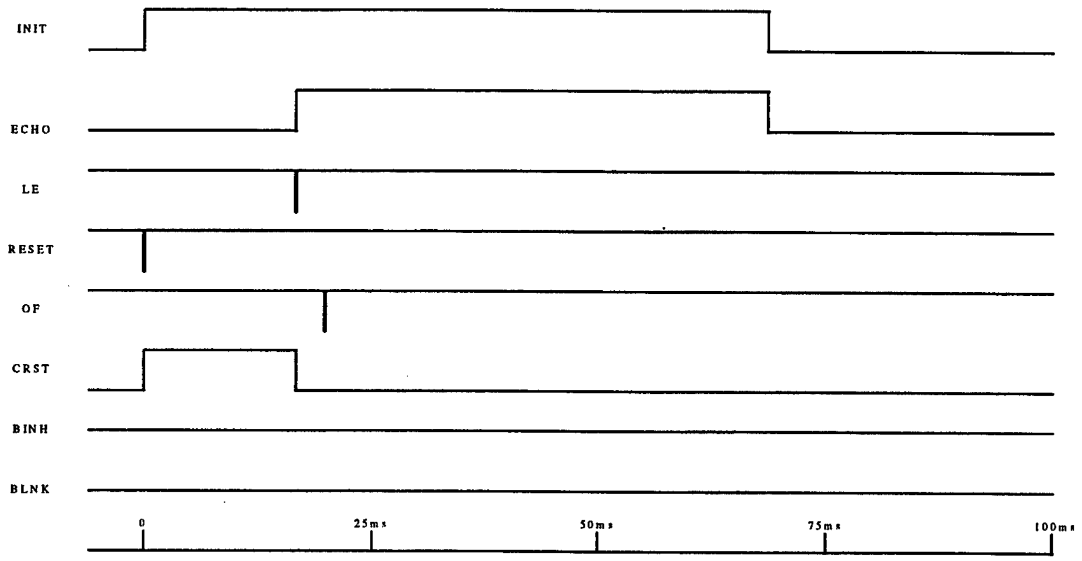

Figure 6. Ranging Controller Timing Diagram. After Ref. [4]

The 125 KHz timer generates an accurate timing signal which is used for running the binary counter. The timer circuit consists of a crystal oscillator and binary counter. The crystal oscillator generates an 8 MHz square wave that ensures accurate and consistent time-to-distance conversions. The crystal oscillator output is sent to a divide-by 64 counter that provides the 125 KHz square wave clock (CLK) to the 8-bit binary counter.

The reference latch controls the operation of the binary counter. It generates Counter Rest (CRST) when either INIT or Counter Overflow (OF) is generated. The eight-bit binary counter utilizes eight stages of a twelve-stage binary counter IC. It counts the number of clock pulses that are sent from the clock circuit during the time between generation of INIT and OF, and outputs the eight-bit binary count to the echo latch. Also, by utilizing **Q3** (the third flip-flop output) as the least significant digit in the count, a divide-by-eight function is executed. OF is generated after **Q11** is set high indicating the maximum eight-bit count (2^8) has been reached. OF commands the reference latch to generate Counter Reset (CRST). CRST causes all output bits to go low pending the start of the next ranging cycle.

The echo latch reads the output of the binary counter and latches the counter output at the time ECHO is generated. When Latch Enable (LE), which is the differentiated ECHO signal, is received, the latch passes the values that appear at its inputs to their respective outputs. They are then sent to the LED driver.

The LED driver conditions the eight-bit latched counter output so that it is able to drive both the LED display and the I/O board input port. The driver reads the output from the echo latch then inverts and outputs the data to the eight-bit LED display and to the I/O board. The driver is powered by the PC and provides the current required to drive the inputs of both devices while isolating the RCC from the I/O board.

The eight-bit LED display is made up of eight individual LEDs that display the output of the ECHO latch. The LEDs are located on the RCC. They facilitate quick verification of proper board operation and serve as a valuable trouble shooting tool. The user should note that since the RCC is connected to the I/O board, if the PC is powered down then the LEDs will all go high. Therefore, the 37-pin connector that interfaces the

RCC to the computer must be disconnected if the ranger is operated independently while the PC is off.

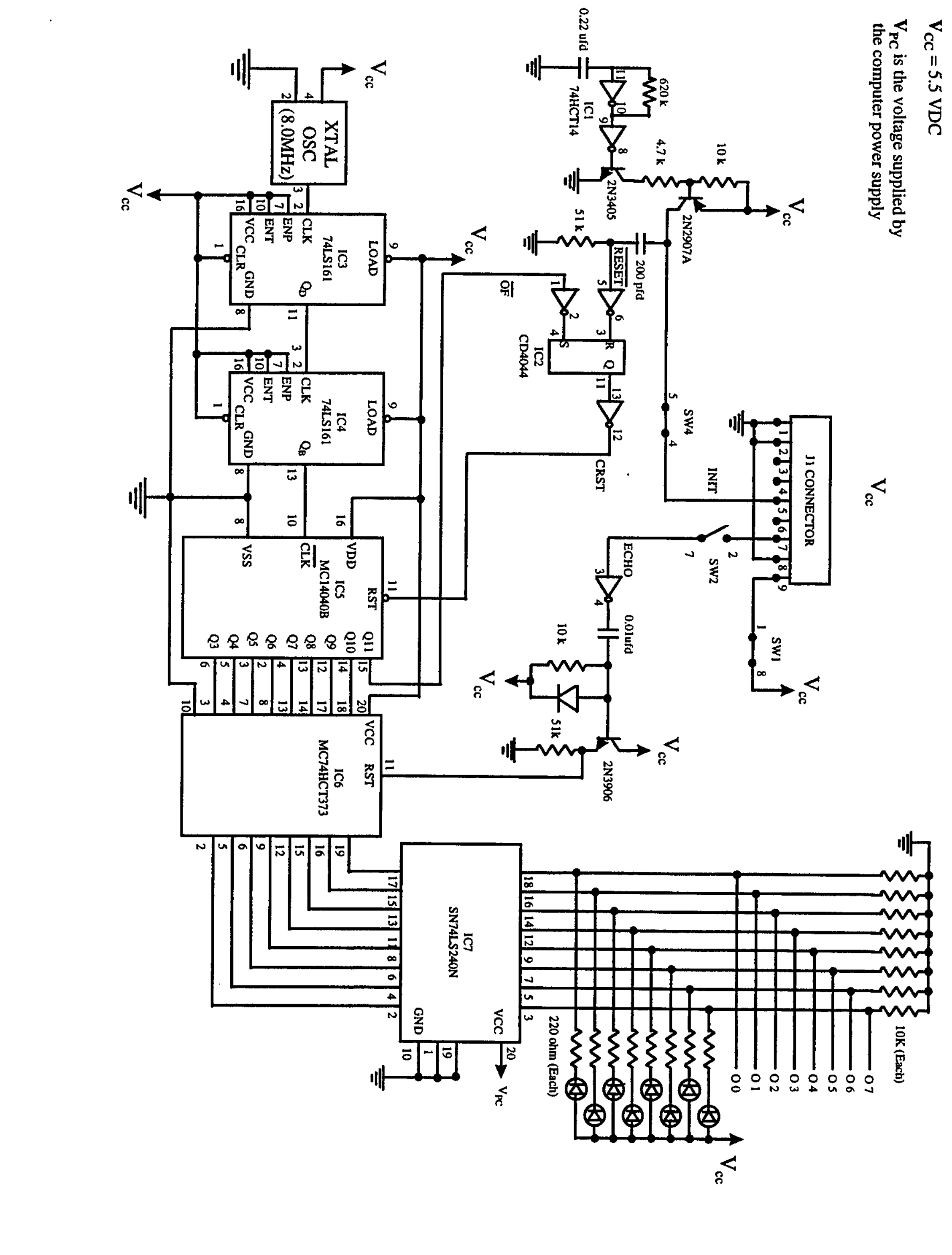

Figure 7. Ranging Controller Schematic

C. DIGITAL INPUT/OUTPUT BOARD

The use of a sonar transducer as a second sensor requires that the Zebra-ZERO PC have on-demand, real-time access to the Ultrasonic Ranging System's ranging data. The device used to provide this link is the CyberResearch CYDIO 24 Digital I/O Board [Ref. 5].

1. CYDIO 24 I/O Board Description

The I/O board supports the input and output of three eight-bit words via three digital I/O ports, as well as limited interrupt generating capability all through a single 37-pin connector [Ref. 5]. The board has three eight-bit ports that can be utilized as either inputs or outputs. They are designated ports **A, B** and **C**. Although not used for this thesis, 2 connector pins are designated for providing limited interrupt servicing capability. The I/O board is installed in a vacant slot on the PC's motherboard. The physical installation provides the board with power, communication, and a data transfer link.

2. CYDIO 24 Implementation

In order for the PC and the I/O board to communicate, a base address is designated and assigned to the I/O board. The base address is the location to which the control software writes and from which it reads when communicating with the I/O board. The base address is set by manipulating an eight-pin dip switch on the I/O board [Ref. 5, p. 3].

The I/O board has two additional functions that may be set on the board but are not utilized in the applications described in this thesis. They are the wait state jumper block and the interrupt jumper block. The wait state jumper block allows the designer to slow the PC down when accessing the board, and the interrupt jumper allows the designer to map the interrupt directly into the PC bus. Their use and implementation may be reviewed in the CYDIO technical manual [Ref. 5, p. 4].

The CYDIO I/O board has many software-generated capabilities that require the utilization of a library of **C** based CYDIO functions, and task specific **C** code. The I/O board comes with a comprehensive library of functions that allow the designer to set up the ports for input or output and to perform I/O functions [Ref. 6]. The code is designed to be integrated into standard **C** programs.

The CYDIO 24 is installed in the Zebra-ZERO PC motherboard slot with the base address set to 300H. A 37-pin cable is plugged into the board and gives it access to the eight-bit ranging data provided by the sonar ranging system via the I/O board's port **A**.

The systems described in this chapter are utilized to implement the control laws that allow manipulation of objects while the manipulator is on a moving base. It is necessary to combine the control algorithms that are used with the force sensor and the sonar ranger into one integrated control algorithm that works to seamlessly impose a control goal on its environment. The design concept that address this complex problem is the *Hybrid Control Approach*, discussed in the next chapter.

III. HYBRID CONTROL

The Hybrid Control Approach [Ref. 7] is used to address the design problem of utilizing both a sonar ranging system and a force sensor to execute pick-and-place algorithms while the manipulator is mounted on a moving platform. By adding the additional sensor to the Zebra-ZERO system, the robot arm is endowed with a greater degree of intelligence in dealing with its environment [Ref. 8]. The greater degree of intelligence is needed to allow the manipulator to perform tasks associated with being mounted on a moving platform. Therefore, a more complex controller must be utilized.

A hybrid dynamic system consists of a reasoner interfacing with a continuous-time system. The discrete event system is a decision maker or controller that operates at a strategic level. The continuous-time system is the plant and its continuous-time controller executing the will of the reasoner.

Hybrid dynamic systems are used in a wide range of applications. For example, hierarchical analysis of manufacturing systems [Ref. 7], and developing optimal dispatching policies for elevator control systems [Ref. 18]. Research on hybrid systems include development of flight control systems [Ref. 19], constrained robotics systems [Ref. 19], and a general basis for the modeling of a wide range of motion control systems [Ref. 20].

Hybrid dynamic modeling uses rule-based process monitoring and discrete event control to move the control state closer to the completion state [Ref. 7]. The programmer designs a reasoner that strategically moves the controller from an initial state to completion. However, the responsibility of controlling the manipulator is not placed on the reasoner. The reasoner calls discrete control processes that are responsible for implementing the continuous-time controller based on the rules imposed by the reasoner algorithm. For example, while the manipulator is moving toward an object in free space using the sonar for range information, the controller uses position control commands to execute the current control law. However, once the end-effector contacts a surface, the sonar feedback is no longer useful. The control law must change to one that uses the force sensor to detect the forces imposed on the end-effector. This requires a different set of Zebra-ZERO movement commands. The algorithm that controls the manipulator's approach to the surface is one task-level controller, and the algorithm that controls the manipulator while it is in contact with the surface is another task-level controller. The reasoner commands the approach controller to begin its task. Once the approach controller detects contact it relinquishes control of the manipulator and passes the state of the control event to the reasoner. The reasoner then decides which discrete event controller is to be used next. In this case, it calls the discrete controller that maneuvers the manipulator while it is in contact with the surface.

The controller also utilizes commands that position the manipulator relative to its current position. The installed software provides a function (*kin.c*) that solves for the current position of the manipulator relative to the base frame. It also provides a function (*ikin*) that solves for the joint angles that would place the manipulator at a desired position. These functions utilize sets of equations that are specific to the Zebra-ZERO. The equations are defined as *kinematic* equations. The solution of the kinematic equations are used to position the manipulator in various modes of operation.

This chapter covers the concepts used to implement hybrid controllers to solve the problem of controlling the manipulator while it is mounted on a moving base. The topics covered are control system overview, plant kinematics, and controller architecture.

A. HYBRID CONTROL SYSTEM DESCRIPTION

A block diagram of the hybrid control system is shown in Figure 8. It consists of a task reasoner, controller, D/A converter, A/D converters, external sensors, internal sensors, power amplifiers, and plant. All elements of the control system work to produce an effect on the environment.

The task reasoner is the decision maker for the control system. The reasoner monitors the state of the controller to decide if the current control law (force, torque, pure position) or control state is valid, and to command the controller to switch to another control law or control state as necessary. The reasoner is a software implemented switching routine that is interwoven into the controller software acting as the strategic part of the controller process.

The controller implements the control law and/or control state selected by the reasoner. The controller consists of both hardware and software working together to perform according to the current control law. Elements that make up the controller include the control functions, PC, A/D converters, power amplifiers, actuators, and both internal and external sensors.

The software side of the controller schedules the control system's actions by observing the process state and making the appropriate control commands to the actuators. The software-driven controller gives flexibility to the robotic system. It allows the designer to customize the controller algorithm according to a particular task.

The controller hardware executes the control process and provides feedback to the controller software. The hardware directly interacts with the environment through the actuators and the plant to produce the desired results.

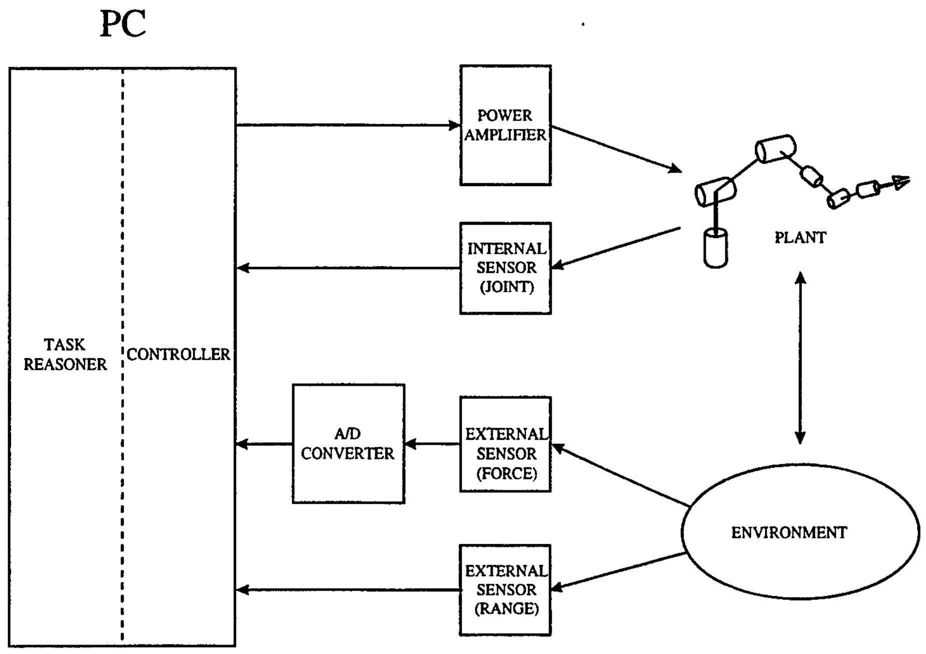

Figure 8. Control System Diagram. After Ref. [7]

The sensors are utilized to report the state of the plant and the environment. The sensors in the hybrid control system are classified as internal state sensors and external state sensors [Ref. 7]. The internal state sensors provide feedback information that relates to the state of the plant in the form of joint angles. The optical position encoders make up the internal sensor system. The external state sensors provide feedback on the state of the environment in the form of a force vector and ranging data. The force sensor and the sonar ranger are the two external sensors used in this study.

The force sensor interfaces to the control software via the A/D converter. The A/D converter, located in the force sensor body, digitizes the analog force and torque signals and outputs them to the PC.

The power amplifiers condition and amplify the control signals sent from the motor controller board in the PC to the motors in the plant. The amplifiers provides commanded current (between 0 and 3 amps) that drive the prime movers in the plant. The plant consists of the joint motors and their associated linkages. The components of the plant work together to exert the control law on the environment.

The environment, as shown in Figure 8, denotes the entity with which the plant interacts. The environment is operated upon by the plant and observed by the external sensors. The end objective of the robot control system is to manipulate the environment to produce a desired final state.

B. PLANT KINEMATICS

Hybrid control movements performed by the Zebra-ZERO control software depend on the solution of the manipulator kinematics. The kinematic equations describe the tool frame relative to the base frame and the base frame relative to the tool frame as a function of a particular robot's joint angles and link lengths [Ref. 3]. In the case of the Zebra-ZERO, the variables consist of six joint angles and six link lengths represented by θ_i and l_i respectively. The mechanics and control of the Zebra-ZERO rely on the solution of two types of kinematic problems. They are the forward and inverse kinematic solutions.

1. Forward Kinematic Solution

Forward kinematics involves solving the static, geometrical problem of computing the position and orientation of the end-effector relative to a particular reference frame [Ref. 3]. In the context of the Zebra-ZERO, it is desired to compute the position and orientation of the tool frame relative to the base frame, given the six joint angles.

The manipulator is equipped with optical position encoders that provide joint position information to the PC . The PC utilizes this information to generate the forward kinematics solution. During manipulator operations, the servo control software is constantly running to generate smooth paths, operate the force sensor, monitor the current position of the arm, and to continuously update the commanded position for each of the joint motors [Ref. 1, p. 7]. Through all of the intensive calculations the control system must always know the location of the tool frame with respect to the base frame. The relationship of the tool frame relative to the base frame is described as a transformation matrix that is a combination of a position vector and a rotation matrix.

The *position vector* represents the tool frame, {B}, relative to the base frame, {A}, and is annotated as $^{\mathbf{A}}\mathbf{P}$. The position vector is simply a vector that describes a point in space. The base of the vector is at the origin of the base frame and its tip is at the origin of the tool frame. The position vector is written as

$$^{A}P = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}, \qquad (1)$$

where the individual elements describe the location of the tool frame in *x*, *y*, *z* coordinates.

The orientation of the tool frame relative to the base frame is described by three unit vectors that give the principal directions of the tool coordinate system relative to the base frame. For convenience, the three vectors are written as one 3×3 matrix called the *rotation matrix*. The rotation matrix may be described in short-hand, unit vector, or matrix notation forms as follows:

$$^{A}_{B}R = [\,^{A}\hat{X}_{B} \;\; ^{A}\hat{Y}_{Bt} \;\; ^{A}\hat{Z}_{B}\,] = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}. \qquad (2)$$

Given the position vector and the rotation matrix described by Equation (1) and Equation (2) , an object described in an arbitrary coordinate frame, $^{\mathbf{B}}\mathbf{P}$, may be described or mapped in the base coordinate frame. This mapping is achieved by describing the position of the origin of the arbitrary frame relative to the base frame described by $^{A}\mathbf{P}$, and by describing the orientation of the arbitrary frame relative to the base frame described by $^{A}_{B}\boldsymbol{R}$. The translational mapping is described by the equation

$$^{A}P = {}^{B}P + {}^{A}O_{B}, \tag{3}$$

where $^{A}O_{B}$ is the location of the origin in the tool frame relative to the origin of the base frame. The rotational mapping is described by the equation

$$^{A}P = {}^{A}_{B}R \; {}^{B}P. \tag{4}$$

Equations (3), and (4) may be combined into one equation that describes both translational and rotational mappings as follows:

$$^{A}P = {}^{A}_{B}R \; {}^{B}P + {}^{A}O_{B}. \tag{5}$$

Figure 9 provides a visual representation of the transformation between coordinate systems.

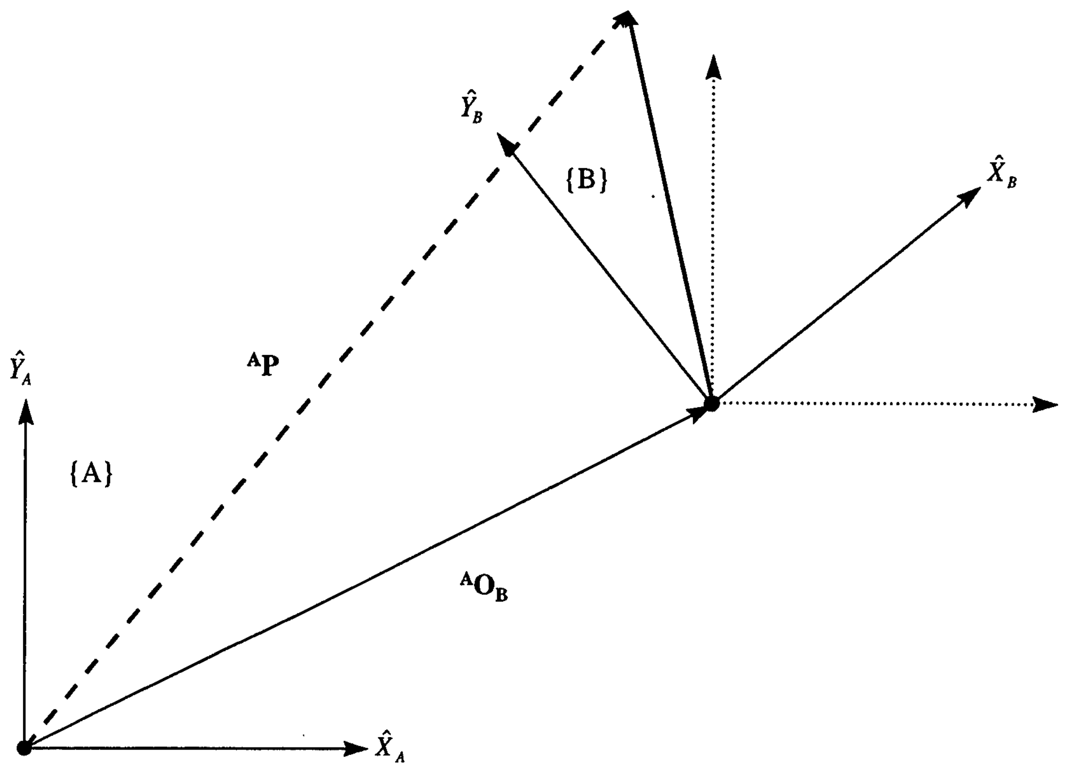

Figure 9. Translation and Rotation Translation. After Ref. [3]

To make the mapping a matrix operation, Equation (5) may be written in compact form as

$$\begin{bmatrix} {}^{A}P \\ 1 \end{bmatrix} = \begin{bmatrix} & {}^{A}_{B}R & & {}^{A}O_B \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{B}P \\ 1 \end{bmatrix}, \tag{6}$$

where a 1 is added as the last element of the 4×1 vectors and the row vector **[0 0 0 1]** is added as the last row of the 4×4 matrix for conceptual convenience. This form is that of a homogeneous transformation matrix [Ref. 3].

Now that the transformation from one coordinate system to another has been made, the same concepts will be used to specify one frame relative to another. The notation used for describing frame {B} relative to frame {A} is ${}^{A}_{B}\mathrm{T}$, where **T** is a transformation operator that rotates and translates a vector **${}^{B}P$** to compute the new vector **${}^{A}P$** [Ref. 3]. The transformation operator is described as

$$ {}^{A}_{B}\mathrm{T} = \begin{bmatrix} & {}^{A}_{B}\mathrm{R} & & {}^{A}\mathrm{O}_{B} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{7}$$

Therefore, equation (6) may be written as

$$ {}^{A}\mathrm{P} = {}^{A}_{B}\mathrm{T}\ {}^{B}\mathrm{P}. \tag{8}$$

Transformation operators may be combined to translate vectors across multiple frames. For example, if the transformation matrices are known across frames {A}, {B}, {C}, and {D}; the transformation of a vector, **${}^{D}P$**, defined in frame {D} to frame {A} is accomplished utilizing the following:

$$ {}^{A}P = {}^{A}_{B}\mathrm{T}\ {}^{B}_{C}\mathrm{T}\ {}^{C}_{D}\mathrm{T}\ {}^{D}P = {}^{A}_{D}\mathrm{T}\ {}^{D}P, \tag{9}$$

where the combined transformation matrix is defined as

$$ {}^{A}_{D}\mathrm{T} = {}^{A}_{B}\mathrm{T}\ {}^{B}_{C}\mathrm{T}\ {}^{C}_{D}\mathrm{T}. \tag{10}$$

This same concept is used to transform the location of the tool frame of the Zebra-ZERO into base frame coordinates. For a six-link robot arm the combined transformation matrix is described as

$$ {}^{0}_{6}\mathrm{T} = {}^{0}_{1}\mathrm{T}\ {}^{1}_{2}\mathrm{T}\ {}^{2}_{3}\mathrm{T}\ {}^{3}_{4}\mathrm{T}\ {}^{4}_{5}\mathrm{T}\ {}^{5}_{6}\mathrm{T} \tag{11}$$

Since the link distances and the angles made between the extensions are known constants imbedded in the Zebra-ZERO software, and the optical encoders provide the joint angles, all of the transformation matrices contain known values. Therefore, the elements of the combined transformation matrix may be computed by multiplying the individual link transforms. The end result of the multiplication is defined as the kinematics of the manipulator.

The kinematic solution used in the Zebra-ZERO is found by utilizing the *forward* or *direct* kinematic method described above in [Ref. 3]. The Zebra-ZERO kinematic solution provided by the Integrated Motions, Inc. [Ref. 1, p. 65] is as follows:

$$ {}^{0}_{6}T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad (12) $$

where

$$r_{11} = c_1 c_{23} (c_4 c_5 c_6 - s_4 s_6) - c_1 s_{23} s_5 c_6 - s_1 (s_4 c_5 c_6 + c_4 s_6),$$

$$r_{21} = s_1 c_{23} (c_4 c_5 c_6 - s_4 s_6) - s_1 s_{23} s_5 c_6 + c_1 (s_4 c_5 c_6 + c_4 s_6),$$

$$r_{31} = s_{23} (c_4 c_5 c_6 - s_4 s_6) + c_{23} s_5 c_6,$$

$$r_{12} = -c_1 c_{23} (c_4 c_5 s_6 + s_4 c_6) + c_1 s_{23} s_5 s_6 + s_1 (s_4 c_5 s_6 - c_4 c_6),$$

$$r_{22} = -s_1 c_{23} (c_4 c_5 s_6 + s_4 c_6) + s_1 s_{23} s_5 s_6 - c_1 (s_4 c_5 s_6 - c_4 c_6),$$

$$r_{32} = -s_{23} (c_4 c_5 s_6 + s_4 c_6) - c_{23} s_5 s_6,$$

$$r_{13} = -c_1 c_{23} c_4 s_5 - c_1 s_{23} c_5 + s_1 s_4 s_5,$$

$$r_{23} = -s_1 c_{23} c_4 s_5 - s_1 s_{23} c_5 - c_1 s_4 s_5,$$

$$r_{33} = -s_{23} c_4 s_5 + c_{23} c_5,$$

$$p_x = c_1 (l_1 c_2 - l_2 s_{23}),$$

$$p_y = s_1(l_1c_2 - l_2s_{23}),$$

$$p_z = l_1s_2 + l_2c_{23},$$

and

$$s_i = \sin(\theta_i),$$

$$c_i = \cos(\theta_i),$$

$$c_{ij} = \cos(\theta_i + \theta_j),$$

$$s_{ij} = \sin(\theta_i + \theta_j).$$

The Zebra-ZERO servo control software is actively running while the manipulator is energized either to position the manipulator in response to a call to a positioning function or to maintain the manipulator in its current position [Ref. 1]. The forward kinematic solution is utilized to provide the position and orientation feedback required to accurately place the tool frame. The forward kinematic solution (*kin.c*) is also available as a utility function to be utilized in general manipulator control algorithms.

2. Inverse Kinematic Solution

Manipulator control not only relies on the ability to compute the current position and orientation of the tool frame relative to the base frame, but it also requires the ability to move the tool frame to a desired configuration and coordinate position relative to the base frame according to the control law. This problem involves solving the inverse kinematics of the manipulator. The problem is posed as the following: Given a desired position and orientation of a tool frame {B} relative to the base frame {A}, calculate a set of joint angles which will attain this position within the work space of the manipulator [Ref. 3].

The solution to the inverse kinematic problem is more involved than the forward kinematic solution. The forward kinematic solution has a unique solution for the given joint angles. However, the inverse kinematic solution has multiple solutions if the desired configuration is within the manipulator's dexterous workspace, and has no solution if it is not [Ref. 3]. For example, a manipulator with six joints may have as many as sixteen possible solutions [Ref. 10]. There are no general algorithms that can produce a solution to this nonlinear problem. However, a special case is engineered into the Zebra-ZERO that produces a closed-form solution.

Piper's solution reduces the number of solutions to the inverse kinematics problem by requiring that the last three axes intersect [Ref. 11]. Piper's solution reduces the number of possible solutions to the last three joint angles to two. Therefore, the number of solutions for the manipulator is twice the number of solutions found for the first three joints [Ref. 3].

The Zebra-ZERO solves the inverse kinematics problem by utilizing an algebraic solution to solve the first three angles then utilizing a geometric solution to solve the last three angles.

The following is the algebraic, closed-form solution to the inverse kinematics. Given **P** is the position vector that describes the desired location of the tool frame origin in the base frame coordinate frame, where

$$\mathrm{P} = \begin{bmatrix} \mathrm{p_x} \\ \mathrm{p_y} \\ \mathrm{p_x} \end{bmatrix}. \tag{13}$$

we may compute the distance from the origin of the base frame to the origin of the wrist frame as

$$r = \sqrt{P' P} . \tag{14}$$

For a solution to exist the inequalities

$$r + 0.1 < l_1 + l_2 \tag{15}$$

and

$$r - 0.1 > l_1 - l_2 \tag{16}$$

must be true. If Equations (15) and (16) are true, the solution is within the dexterous workspace of the manipulator. Next, the distance r_{xy} projected into the x-y plane is computed as

$$r_{xy} = \sqrt{p_x^2 + p_y^2}\ . \tag{17}$$

Let

$$\alpha = Tan^{-1}\left(\frac{p_z}{P_z}\right), \tag{18}$$

$$\beta = Cos^{-1}\left(\frac{l_1^2 + r^2 - l_2^2}{2rl_1}\right), \tag{19}$$

$$\gamma = Cos^{-1}\left(\frac{l_1^2 + l_2^2 - r^2}{2l_1l_2}\right). \tag{20}$$

Using **P** and Equation (18) through Equation (20), the first three joint angles are solved as follows:

$$\theta_1 = Tan^{-1}\left(\frac{p_y}{p_x}\right), \tag{21}$$

$$\theta_2 = \alpha + \beta, \tag{22}$$

$$\theta_3 = \gamma - 1.5\pi . \tag{23}$$

Next, the last three joint angles are determined geometrically. The unit vector normal to the plane of the arm is calculated as

$$\hat{n} = \begin{bmatrix} \sin(\theta_1) \\ -\cos(\theta_1) \\ 0.0 \end{bmatrix} . \tag{24}$$

Then, the projected vector along the **J4** axis is solved as

$$q = \begin{bmatrix} \cos(\theta_2 + \theta_3 + \frac{\pi}{2})\cos(\theta_1) \\ \cos(\theta_2 + \theta_3 + \frac{\pi}{2})\sin(\theta_1) \\ 0 \end{bmatrix} , \tag{25}$$

then normalized to

$$\hat{q} = \frac{q}{\sqrt{q'q}} \tag{26}$$

the unit vector along the **J4** axis. The unit vector normal to $\hat{n}$ and $\hat{q}$ is then calculated as

$$\hat{s} = \hat{n} \otimes \hat{q} . \tag{27}$$

Next, x^* and y^*, the wrist z-axis projections on $\hat{s}$ and $\hat{n}$ respectively are computed utilizing the equations

$$x^* = {}^t\hat{Z}'_A \ \hat{s} , \tag{28}$$

and

$$y^* = {}^t\hat{Z}'_A \ \hat{n} . \tag{29}$$

Where ${}^{t}\hat{Z}'_A$ is the unit vector in the direction of the z-axis in the base frame. Angles θ_4 and θ_5 are calculated as

$$\theta_4 = Tan^{-1}\left(\frac{y^*}{x^*}\right) \tag{30}$$

and

$$\theta_5 = \mathrm{Cos}^{-1}({}^{t}\hat{Z}'_A\ \hat{q}). \tag{31}$$

Finally, the unit vector pointing along the **J5**-axis is calculated as

$$\bar{n} = \begin{bmatrix} \cos(\theta_4)\hat{n}_1 - \sin(\theta_4)\,\hat{s}_1 \\ \cos(\theta_4)\hat{n}_2 - \sin(\theta_4)\,\hat{s}_2 \\ \cos(\theta_4)\hat{n}_3 - \sin(\theta_4)\,\hat{s}_3 \end{bmatrix}. \tag{32}$$

The angle θ_6 is calculated using Equation (32) and the two unit vectors pointing in the direction of the x and y-axes, ${}^{t}\hat{X}_A$ and ${}^{t}\hat{Y}_A$ respectively, in the base frame coordinate system as follows:

$$\theta_6 = \mathrm{Tan}^{-1}\left(\frac{-\bar{n}'\ {}^{t}\hat{X}_A}{-\bar{n}'\ {}^{t}\hat{Y}_A}\right). \tag{33}$$

The Zebra-ZERO servo control software utilizes the inverse kinematic solution to position the manipulator in response to calls to positioning functions or to maintain the manipulator in its current position [Ref. 1]. The inverse kinematic solutions are utilized to provide the joint angles required to place the tool frame in a desired position and orientation based on the control solutions calculated by the control software. The inverse kinematic solution (*ikin.c*) is also available as a utility function to be utilized in general manipulator control algorithms.

C. CONTROLLER ARCHITECTURE

Because the control environment utilizes a force sensor and a sonar ranger, the controller cannot employ a single generic control law to finish a process. Therefore, the manipulator control model is built according to the process properties described by each discrete controller. The hybrid controller architecture illustrates how the software is used to implement multiple control laws. The controller architecture models the control law in phases as a state diagram [Ref. 7].

Figure 10 displays an example of a typical software controller model. The elements of the controller are the reasoner, phase controllers (**PC1** - **PC3**), and subphase controllers (**SPC1** - **SPC2**). Each element of the controller model is a software implemented task. The control objective is accomplished by programming the reasoner to plan and schedule the order of events that will accomplish the control goal. Individual events are represented by a phase controller. The phase controller (**PC**) is responsible for executing a complex portion of the control process. For example, a phase may consist of picking up an object while another might be moving the object from one location to another. Each phase controller may be composed of subphase controllers (**SPCs**). The **SPCs** are the more rudimentary functions that aid in accomplishing the task assigned to a particular **PC**. **SPCs** may perform tasks such as acquiring sonar ranging information or executing pure position control functions such as open-loop manipulator positioning.

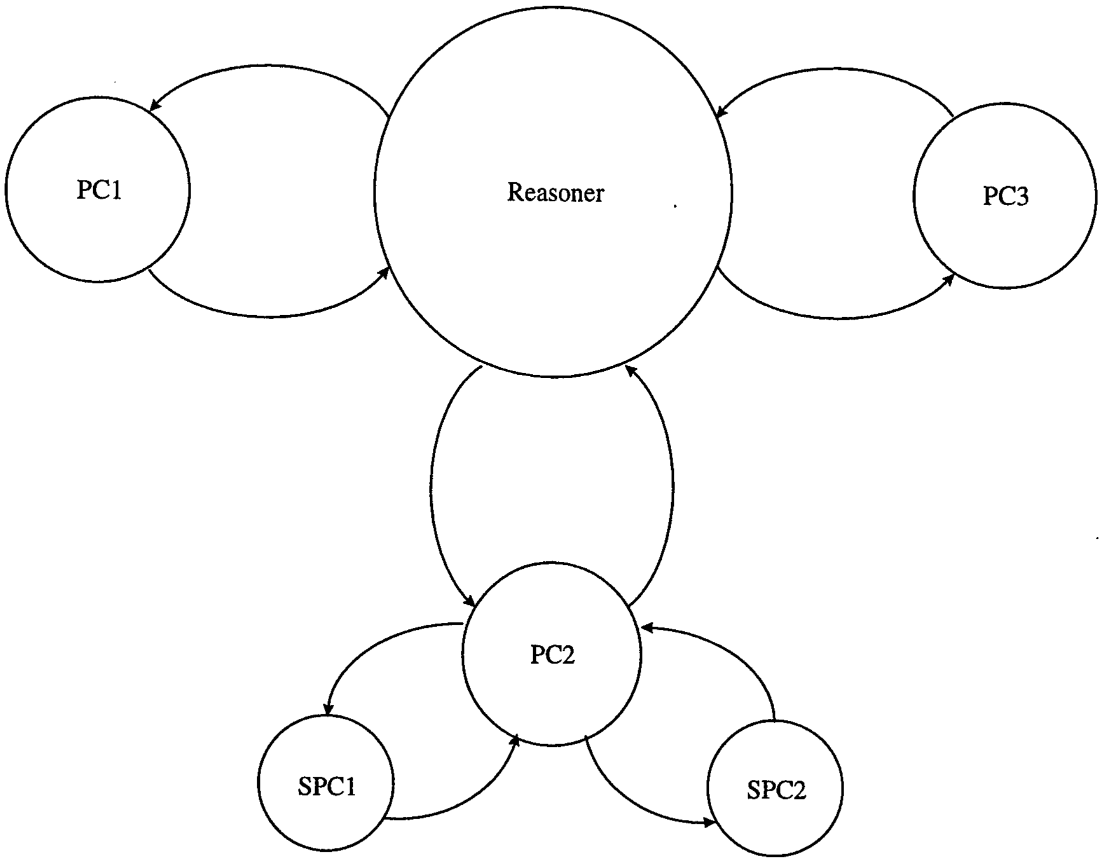

Figure 10. Software Controller Model

Transitions between processes are governed by the state or condition of the plant and environment. These conditions are defined as maintaining, enabling, and disabling conditions [Ref. 9]. Maintaining conditions are observed states that maintain the controller in the current process. Enabling conditions are observed states that cause the controller to transition to another state. Disabling conditions impose a constraint on the controller such that if a particular plant state is encountered the process or control event is terminated.

Design of a software controller utilizing the architecture described above is a top-down process that enables the programmer to concentrate on one process at a time. The steps of the design process are as follows:

- Establish an overall control goal and define the sequence of events to be coordinated by the reasoner.
- Develop events that bring the current state closer to the desired state and define them as processes, subprocesses, and process transitions.
- Implement the reasoner, processes, subprocesses, and process transitions in a complex hybrid controller code as programs and functions.

The hybrid control approach described in this chapter is the design concept used to implement the control algorithms that enable the Zebra-ZERO system to execute various control objectives while it is mounted on a moving platform. The control algorithms utilize functions contained in the Zebra-ZERO software library, integrated in task-specific code, to control the manipulator. Chapter IV introduces the concepts used to develop and execute manipulator control algorithms.

IV. ZEBRA-ZERO SOFTWARE OVERVIEW

The Zebra-ZERO software package includes a library of compiled robot control functions, demonstration programs, and a Borland Turbo C++ compiler. The source code for the demonstration programs are provided to give some insight into writing the **C** code for the Zebra-ZERO. For the novice Borland **C++** user, a first attempt at developing a running program may be quite challenging. However, once the basic methodology of developing a project file is understood, the user may utilize the Zebra-ZERO function library to develop clever programs and original functions.

A. TURBO BORLAND C++ OVERVIEW

The Zebra-ZERO code is written using Turbo Borland **C++**, version 3.0. The intent of this section is to provide the programmer with enough basic information to start programming in the Borland environment with the least amount of initial time investment. It is expected that the programmer has some background in **C** programming. It should be noted that, although Turbo Borland **C++** has **C++** programming capability, *robot.lib* may only be accessed utilizing the **C** compiler. Kernigham and Ritchie [Ref. 12] and the Turbo **C++** User's Guide [Ref. 13] are excellent technical references for programming in **C** and utilizing the Borland software respectively.

The Borland **C++** software is launched by typing **TC** at the DOS prompt within the ROBOT directory. The Borland programming window, is defined as the Integrated Development Environment (IDE). It contains menus at the top and bottom of the screen with a gray working field. The IDE contains all that is needed to write, edit, compile, and debug a program [Ref. 13].

1. Opening an Existing Project

To appreciate the project format utilized by the Borland software, the programmer must first understand the purpose of the project file. The IDE places all information required to build and run a program into a binary project file whose extension is **.PRJ**. The project file contains the settings for the compiler, linker, make, and librarian options as well as the directory paths, lists of all files that make up a project, and the Turbo Assembler translator [Ref. 13, p. 34]. The Borland compiler automatically takes all the programs and files in the project window and compiles, links, and creates the executable file. Therefore, constructing a project file in the IDE frees the programmer from the administrative tasks of constructing and modifying the configuration file used to build the programs defined in a project file [Ref. 13].

Opening a project file allows the programmer to view and edit existing files, and to create new files. To open an existing project, select the following menu items from the top of the IDE:

Project | Open Project.

For instructional purposes, select ARMTEST.PRJ from the *Open Project* dialog box.

After opening the project file, additional windows may or may not be displayed. The most important window for the moment is the Project window. It may appear at the bottom fourth of the IDE. If the Project window does not appear at the bottom of the IDE, select

Window | Project.

The ARMTEST.PRJ project window is then displayed at the bottom of the IDE. The window should list the files *armtest.c* and *robot.lib.*

Double-clicking on *armtest.c* will display the executable source code. The source code may be edited and modified from this screen. However, double clicking on *robot.lib* will display a compiled code that is useless to the programmer. All Zebra-ZERO functions are compiled in *robot.lib* with no source code provided.

As an introduction to the capabilities of the Borland window, the programmer is invited to execute this code from within the Borland environment. To execute this program select

Run | Run.

The program should run as if it were executed from the DOS prompt and upon completion return to the IDE.

2. Creating a Project file

In the following example, ARMTEST.PRJ is used as the model for creating a project file. First, create a new project file name by selecting

Project | Open Project.

Type a project name of choice in the *Open Project* field. For example, the following would be acceptable:

MYTEST.PRJ.

After typing a project name press enter. A project screen with the project name will be displayed at the bottom of the IDE. Next, from the bottom of the IDE, select

Add.

A list of all the **C** coded programs are displayed in the window. If not type

*.C

in the *Name* line and press **Enter**. From the list of **C** source codes select *armtest.c* and select **OK**. The program name *armtest.c* should appear in the project window. Using the same procedure add *robot.lib* to the project menu.

Next, critical default settings must be set to ensure that the files are properly compiled and linked. This is done by opening the *Code Generation* window and setting the parameters as follows:

Selecting the following menu items in the IDE:

Options | Compiler | Code Generation.

Ensure **Large** is selected from the *Memory Model* dialog box, **Default for memory model** is selected form the *Assume SS Equals DS* dialog box, and **Treat enums as ints** is selected from the *Options* dialog box. Then, select **OK** .[Ref. 13]

Next, Turbo C++ must compile and link all the files in the project. This is done by selecting

Options | Build All.

With the project compiled and linked the programmer now has an executable program. To test the project select

Run | Run.

The project may also be executed at the DOS prompt within the ROBOT directory. For example, for the project file MYTEST.PRJ, the programmer types MYTEST then presses **Enter** at the DOS prompt. The program *mytest.c* should perform the identical function as *armtest.c*.

B. EXECUTING FACTORY-PROVIDED PROGRAMS

The Zebra-ZERO has several ready-to-run programs that are included in the ROBOT directory of the PC hard drive. There are programs that execute system tests, program demonstrations, and function familiarization. The computer automatically enters the ROBOT directory at startup. The programs are run by typing the executable and pressing **Enter**.

1. Testing the Zebra-ZERO System

The program *armtest.c* is utilized to test the operation of the Zebra-ZERO system. However, it is recommended that the first-time user execute ARMTEST to obtain a basic working understanding of the Zebra-ZERO. The program walks the user through some basic operational characteristics of the system. Furthermore, the source code is an excellent example of a typical Zebra-ZERO interactive algorithm.

The program starts out by allowing the user to test the operation of the force sensor. The user is prompted to apply forces and moments to the gripper. A bar-graph presentation displays the magnitude of the applied forces/moments.

After the force sensor is tested, the user is prompted to proceed to the next display where the individual joint encoders are tested. All six joint angles as well as a matrix containing the kinematic solution for the location of the tool frame relative to the base frame are displayed.

Finally, the arm is tested through various dynamic movements. The user is prompted to prepare the arm for maneuvering out of its nest (support bracket) and for the power switch to be depressed. The arm then moves to the ready position where it awaits the user's acknowledgment to demonstrate movements of all axis motors and the gripper. After that, it returns to the nest.

2. Maneuvering In and Out of the Nest

There are two programs provided that maneuver the arm in and out of the nest. They are *homerobot.c* and *backhome.c*. Both may be executed at the DOS prompt within the robot directory. The source codes are short and execute one specific task.

The program *homerobot.c* maneuvers the manipulator from the nest to the ready position exclusively. The arm must be in the nest before the program is executed or the arm will shut itself down after attempting to find the nest.

Executing *backhome.c* returns the arm to the nest from any position within the robot's workspace. This command is best utilized during program development and debugging. It gives the user the ability to leave the arm in a position for analysis at the end of a program. The user may then use *backhome.c* to return the arm to the nest for the next run.

3. Utilizing the Command Shell

Executing the command **INTERACT** at the DOS prompt in the Robot directory initiates the Interact command shell. The shell allows the user to execute a number of the Zebra-ZERO library commands, available in *robot.lib*, using single line commands. The user may use the shell as a learning tool to gain understanding of some of the more cryptic control functions. It may also be used as a program development and planning tool.

4. Reading Arm Status

The program *status.c* is a valuable programming and motion planning tool. It displays the manipulator's current status and control settings. This information may be used for determining approximate joint angles or for establishing the tool frame that places the manipulator at a desired location or orientation for use within a program. It can only be called when the arm is in a static condition. However, the manipulator does not have to be energized for the program to return data. Executing *status.c* displays the position and orientation of the tool frame relative to the base frame, the joint angles, the status of the computer generated flags, and the status of the user generated flags. All information is displayed on a single screen display.

C. IMPLEMENTING POSITION CONTROL MODE OF OPERATION

1. Description

Position control mode is the traditional mode of operation for commercial robots. In this mode the controller positions the manipulator in a specific coordinate position according to the control law in affect. The trajectory of the arm is specified by a series of *via points* through which the arm passes, and a final goal point where the arm will come to rest. Although position control assumes that the manipulator is not constrained, it can be used to position the manipulator in a partially constrained environment where only slight forces are imparted on a rigid surface. The following examples demonstrate these concepts.

2. Demonstration of Position Control Mode of Operation

The program *a_test* demonstrates the operation of the Zebra-ZERO in pure position control mode of operation. It utilizes commands that position the manipulator according to a specified 6x1 joint vector, or as a Cartesian frame locating the tool frame relative to the base frame. The program moves the manipulator through its workspace utilizing the following position control commands:

cmove
hjog
jjog
jmove
jog
sjmove

Details of the operation of these function may be found in the Zebra-ZERO operation manual [Ref. 1]. The source code for *a_test* is listed in Appendix A.

3. Example Program Utilizing Force Threshold Sensing

The program *contour.c* utilizes force threshold sensing and position control to guide a tool. such as a pencil, along a contoured surface. The program demonstrates the Zebra-ZERO's ability to implement the force sensor to detect changes in the environment and adjust the tool frame according to the detected changes using position control. The source code is listed in Appendix A.

The program *contour.c* starts out by placing the manipulator in the ready position and then opening the gripper. The user is then prompted to place the tool in the open fingers of the gripper and press **Return**. The gripper fingers then close on the tool and pauses for the user to indicate that the tool was properly placed. Next, the routine positions the manipulator so that the base plane (**J1** axis) is at -90° and the gripper is positioned 15 cm from the mounting base and 15 cm above the plane containing the mounting base. The tool is then slowly lowered to contact the surface. It is recommended that the surface be slightly compliant to reduce the risk of damaging the manipulator. The tool is incrementally lowered toward the surface at a rate of approximately .1 cm/sec. After each movement, the force sensor is polled to determine if the surface has been detected. When the surface is detected, the manipulator's descent is terminated. The controller then moves the tool frame forward relative to the base frame. After each incremental move the controller polls the force sensor to detect if the force applied to the tool is within a predetermined threshold. If the force is too small the tool is lowered. If the force is too large the tool is elevated. If the measurement is within the threshold, the tool is moved forward. This process is repeated throughout the execution of the program allowing the tool frame to comply to the contour of the surface as it is moving forward. The task terminates after the manipulator has traveled approximately 15 cm. The manipulator returns to the ready position, relinquishes the tool when prompted, then returns to the nest.

D. IMPLEMENTING FORCE CONTROL MODE OF OPERATION

1. Description

Force control mode of operation allows the programmer to utilize the Zebra-ZERO's force control sensor to execute a task. It is implemented using a specific set of commands available in *robot.lib.*

Force control mode utilizes the force sensor combined with the end-effector to exert a specified force on the environment. The exerted force is independent of the position of the end-effector and relies on the programmer-specific force/torque vector. This mode is intended to be used when the end-effector is constrained so that in cannot move freely in space [Ref. 1]. It should also be noted that the manipulator will only comply to forces exerted on the tool side of the force sensor.

The following Zebra-ZERO library commands are utilized in force control mode of operation:

zero_force
set_bias_force
set_damping
set_force_threshold
set_stiffness
stiffness_off
push_with_bias

2. Example Program Utilizing the Command *push_with_bias*

In order to command the manipulator to apply a user-defined force, a standard sequence of commands must be issued. Their order of execution is irrelevant with the exception that *push_with_bias* must be executed last. When *push_with_bias* is executed the robot is placed in pure force control mode of operation.

The function *push_with_bias* has only one argument. It is the duration that the specified force is to be applied in seconds. However, *push_with_bias* utilizes other functions that are designed to govern the *push_with_bias* force control behavior.

The function *set_damping* designates a damping constant for all force controlled motions [Ref. 1, p. 38]. The input argument, *damping*, is a floating-point number between 0.0 and 1.0. The maximum damping occurs when the damping value is set to 0.0. Unstable motions warrant lowering the damping value while sluggish motions warrant increasing the value.

The function *set_stiffness* is used when the Zebra-ZERO is in stiffness control mode of operation, a subset of force control mode. Stiffness control is used to control both position and force at the same time while the arm is unconstrained in the force control mode [Ref. 1]. The programmer is able to specify the desired behavior of the end-effector in the tool coordinate axes. A 6x1 force vector is used to assign stiffness along the tool coordinate axes as well as about their rotational axes. For example, if it is desired to move the end-effector along a surface of which the contour is unknown, the stiffness may be set so that the end-effector is compliant along the Z-axis and stiff to applied forces along all other axes and along all rotational axes. Therefore, *set_stiffness* will control the deflection of the end-effector while it is moving through space while in stiffness control mode which is used when executing *push_with_bias*.

The function *set_force_threshold* is utilized to set the parameters that protect the arm in the event of unexpected collisions, or as a safety feature to protect object(s) being operated upon by the end-effector [Ref. 1, p. 38]. If the magnitude of any force/torque exceeds the corresponding value in the threshold vector, the current motion is aborted. The function's input is a six element force vector that describes the maximum forces and moments that may be applied to the end-effector. The values implemented by calling *set_force_threshold* are utilized in every Zebra-ZERO function. The system is constantly checking for forces/torques that exceed the designated envelope. Although the programmer may explicitly assign the values contained in the threshold vector, the Zebra-ZERO software will not let the force vector exceed the default values that are factory installed in *robot.lib*. The default values are defined as **MIN_THRESHOLD** within the library.

The function *set_bias_force* is used to set the value of the force and the moments that are applied at the end-effector. The input is a six-element vector that describes the magnitude of the forces and moments that will be applied in force control mode of operation.

The force sensor detects the sum of all forces and torques that are applied to it. Therefore, it will superimpose the forces that are applied by a tool or object that is handled by the end-effector as well as the force due to the mass of the end-effector. Furthermore, these forces/torques, which are gravity and mass dependent, change with the orientation of the end-effector. In most cases it is not desirable to include the superposition of the torques/forces in the measurement of the applied forces. The function *zero_force* reads the force sensor to establish an offset from which all subsequent forces are relatively measured [Ref. 1, p. 33]. The offset value is set in the current tool coordinates. Good engineering practice demands that as the tool coordinates change, subsequent calls to *zero_force* are made.

Finally, after all the previously mentioned functions have been implemented, the call to *push_with_bias* can be made. Execution of *push_with_bias* results in the application of the force vector defined in *set_bias_force* with the parameters set by calls to *zero_force*, *set_damping*, *set_force_threshold*, and *set_stiffness*. Implemented correctly, *push_with_bias* can apply a desired force in a vector-defined direction or in the direction of the moments about the coordinate axes in the tool coordinate frame. An understanding of these functions are critical to utilize the force control mode of the Zebra-ZERO.

The program *force.c,* contained in Appendix A, implements the key force control functions and allows the user to experiment with various force control parameters. The algorithm first moves the arm to the ready position. Next, it sets the initial force threshold, stiffness, bias, damping and push time. It then gives the user an opportunity to change the stiffness and damping vectors as well as the bias force and the amount of time the force is applied. After all parameters have been entered, *push_with_bias* is executed. The interactive portion of the program is iterative and allows the user to keep or change parameters each time it is run. When the user is finished, the program may be exited when prompted.

E. SONAR RANGING SYSTEM IMPLEMENTATION

The sonar ranging system is necessary to position the manipulator in close proximity of the object to be manipulated. The sonar ranging data is read by the control algorithm and is used to position the manipulator utilizing position control commands.

The program *ranger.c*, listed in Appendix A, is a simple program that extracts the 8-bit ranging information from the digital I/O board and displays the computed range on the monitor. First the program configures the I/O board to input data from port **A**. Then it reads the information available at port **A** and logically inverts it. The inversion is executed because the ranging data is sent to the I/O board as negative logic. Next, the information is converted to centimeters and is displayed on the monitor. The binary representation of the range data is also displayed. The readings are taken continuously until the user presses a key to end the program.

The Zebra-ZERO programming software may be utilized to structure a broad range of control algorithms. The following chapter describes the algorithms that implement these control concepts in hardware simulations of the Zebra-ZERO system mounted on an autonomous vehicle.

V. HARDWARE SIMULATIONS ON A MOBILE BASE

The integration of the Sonar Ranging System with the Zebra-ZERO Force Control Robot facilitates hardware simulations of a manipulator that must interact with its environment while it is mounted on an autonomous vehicle. The mobile vehicle is simulated by utilizing a cart as the mobile base. The integrated manipulator system mounted on the cart is referred to as the *manipulator platform*. The sonar provides the range data that allows the manipulator control algorithm to detect the target and then approach it without damaging the manipulator platform or the target. Once the sonar sensor has guided the manipulator to where it contacts the target object, the controller utilizes the force sensor to operate on the target.

Controlling the manipulator while it is mounted on a moving base requires the use of software that is able to switch between force and position control laws to accomplish the ultimate control goal. Two project files are presented that test the feasibility of implementing the Zebra-ZERO system on a moving platform. CONTROL1.PRJ utilizes algorithms that test the ability of the manipulator platform to interact with a target object assuming that the target object is stationary. HYBRID2.PRJ utilizes algorithms that test the ability of the manipulator platform to interact with a target object assuming that the target object is in motion.

A. STATIONARY TARGET HARDWARE SIMULATION

1. Controller Description

CONTROL1.PRJ tests the ability of the Zebra-ZERO to perform a complex pick-and-place task where the manipulator is on a mobile delivery platform and the target object is stationary. This type of control algorithm may be adapted to perform tasks in a land-based environment. Variations of this algorithm could be used for ordinance disarmament, chemical spill evaluation and cleanup, and other related tasks.

The project file CONTROL1.PRJ executes a control task that entails grasping a peg located in a hole and placing it on a flat surface in front of the hole. The physical configuration of the peg and the hole is known by the control algorithm, but their exact location is not. The following constraints and assumptions are in effect:

- The manipulator is mounted on a cart, referred to as the *delivery vehicle.*
- The delivery vehicle may be moved during runtime to place the peg and hole assembly within the manipulator's work space. However, the delivery vehicle must be stationary before the peg extraction sequence begins through program termination.
- The sonar transducer is mounted on the manipulator's rotating base carriage and is centered 24 cm above the mounting surface, 7 cm forward of center and 7 cm right of center.
- The hole assembly is a square cylinder that is 3 inches high, with 2 inch sides. The hole is centered at the top of the cylinder.
- The peg is initially located in the hole assembly with its exposed portion consisting of a 1 inch cube, and its hidden portion consisting of a round cylinder 1 inch in diameter extending 1.5 inches into the hole.
- The approximate bearing of the hole assembly is known. Practically, the bearing could be fed to the control algorithm from a video camera.

The Control algorithms used in CONTROL1.PRJ are *control1.c* which is the reasoner, and *phase1.c* which contains the phase and subphase controllers. The source codes are listed in Appendix B.

The reasoner is responsible for managing the phase controllers listed in *phase1.c* and the control flow is modeled in Figure 11. Execution of *control1.c* proceeds as follows:

First, the manipulator is placed in the ready position by executing **PC1**. After which, **PC2** is immediately executed placing the manipulator at a pre-programmed bearing and configuration to facilitate a safe approach to the target. **PC2** then repeatedly calls **SPC1** to acquire the range to the peg. **PC2** commands the manipulator to close the distance to the peg and hover 10 cm above it. The platform must be static before the peg may be removed. Therefore, **PC2** checks that the distance between the platform and the peg has been steady for 8 seconds before attempting to remove the peg. If the platform does not stabilize after a programmed number of attempts, **PC2** terminates and the reasoner executes **PC4** which places the arm back in the nest. If the manipulator is able to successfully place the gripper and the platform is steady, **PC2** terminates and the reasoner calls **PC3** which positions the gripper 1 cm in front of the peg. **PC3** then calls **SPC2** which slowly moves the gripper forward to contact the peg. If contact is not detected, **SPC2** and **PC3** terminate. The reasoner will then call **PC4**. If contact is made, **SPC2** saves the exact location of the peg and terminates. **PC3** then executes **SPC3**. **SPC3** controls the task of extracting the peg from the hole. If the force sensor does not detect that the peg was extracted, **SPC3** and **PC3** terminate, then the reasoner calls **PC4**. If the peg is extracted from the hole, **SPC3** terminates and **PC3** calls **SPC4** which places the peg in front of the hole assembly. **SPC4** terminates when the force sensor detects that the peg has been placed in front of the peg assembly. After which, **PC3** terminates and the reasoner calls **PC4**.

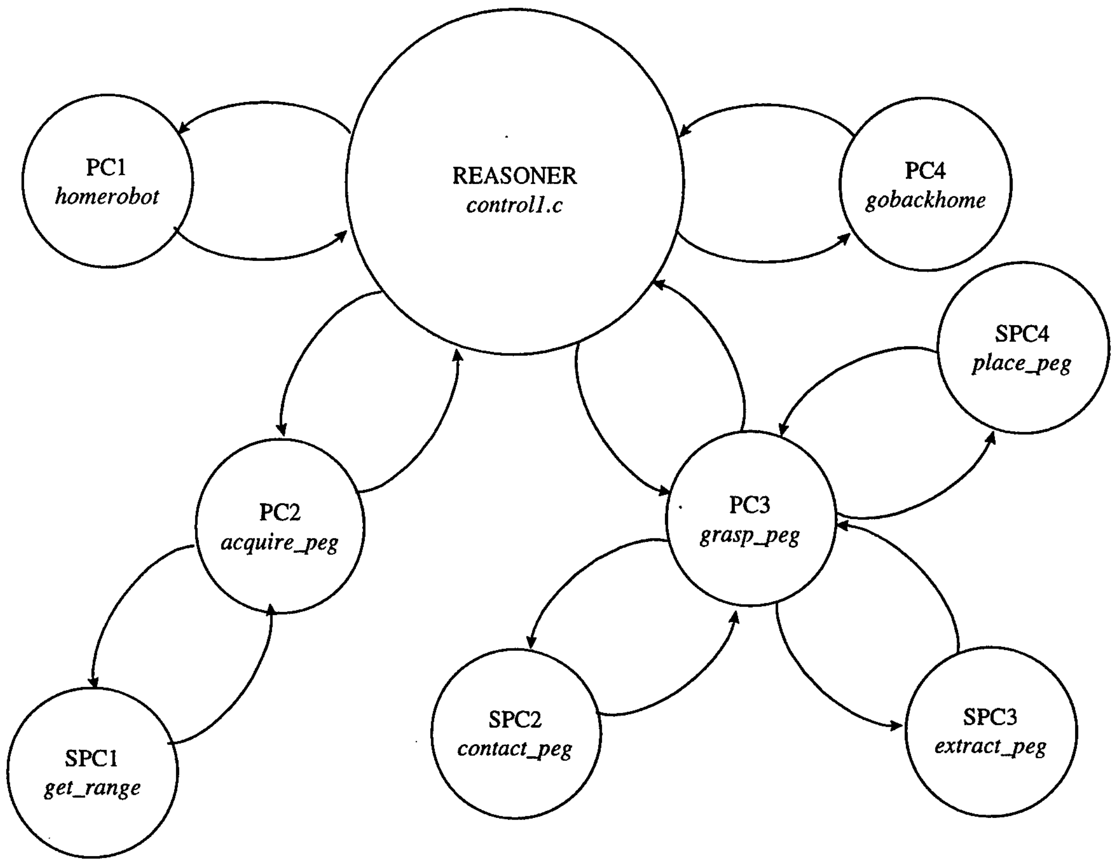

Figure 11. Program *controll.c* Control Model

2. Results

After making several adjustments to the controller, it was able to consistently command the manipulator to retrieve the peg. As the manipulator platform was pushed toward the peg the controller was able to effectively position the gripper so that it maintained its position 10 cm above the peg. Perturbations imposed on the manipulator platform revealed that the controller was able to reposition the gripper so that it remained above the peg.

During early testing, the manipulator displayed sporadic gyrations while making its approach to the peg. The magnitude of these gyrations were on the order of 10 cm and occurred approximately every 10 to 15 sonar readings. Since three range measurements are taken per second, the gyrations were frequent enough to cause the program to abort 50% of the time. The gyrations were caused by sporadic noise experienced by the sonar ranging system. Since the noise was infrequent, the problem was solved by first requiring SPC1 to provide a new sonar reading after a 3 ms if a large change in range was detected. Second, the average of the five most recent measurements was used as the error signal. The second reading reduced the occurrence of the gyrations to approximately every 200 to 300 measurements. Averaging the five most recent measurements smoothed the magnitude of the gyrations to less than 1 cm.

During testing, the sonar ranging data began to experience serious noise. Investigation of the problem revealed that the noise was caused by the system that drives the turret (**J1** axis). The transducer experienced severe interference when placed within a foot of the manipulator. Additionally, the noise only occurred if the turret motor was enabled. After consulting with Integrated Motors Inc. technical support, it is suspected that the turret motor power amplifier was thermally stressed and was slowly degrading. During the degradation period before catastrophic failure, it is possible that the power amplifier emits components of the PWM control signal. The problem is expected to be solved by replacing the power amplifier card.

It was also noted that the performance of the gripper was slowly degrading. Symptoms of this problem included intermittent gripper failure and weakened motor torque. However, replacing the gripper power amplifier should solve this problem.

B. MOVING TARGET HARDWARE SIMULATION

1. Controller Description

CONTORL2.PRJ tests the ability of the Zebra-ZERO to perform a complex pick-and-place task where the target on which the device is being placed is in motion. This type of control algorithm may be adapted to perform tasks in an underwater environment where a hermetically sealed version of the Zebra-ZERO is mounted to a delivery vehicle. The manipulator may be tasked with neutralizing mines, cleaning or inspecting the hull of ships, and many other similar applications.

The project file CONTROL2.PRJ executes a control algorithm that grasps a device whose location and configuration is known by the control algorithm and places it on a vertical surface where the distance between the manipulator and the surface may be constantly varying. The following constraints and assumption are in effect:

- The manipulator is mounted on a cart, referred to as the *delivery vehicle.*
- The delivery vehicle is considered to be part of the manipulator structure for this discussion. The manipulator/mobile cart platform is referred to as the *manipulator platform.*
- The sonar transducer is located on the delivery vehicle 60 cm below the base of the manipulator and is stationary relative to the manipulator.
- The initial location of the object to be placed, referred to as the *device*, is on the platform making it stationary relative to the manipulator, and it is within the manipulator's workspace.
- The surface on which the device is to be placed must be vertical and is referred to as the *target.*
- The device utilizes a magnet-and-spring mechanism that requires a force to be applied to the device while it is placed on the target in order to affix it to the target.
- The target must be ferrous.
- The sonar transducer only provides range information for axial movement. Lateral and vertical movements are not detected.

The Control algorithms used in HYBRID.PRJ are *hybrid1.c* which is the reasoner and *phase2.c* which contains the phase and subphase controllers. The control algorithm is modeled in Figure 12 and the source code is listed in Appendix C.

The reasoner is responsible for managing the phase controllers contained in *phase2.c* and is executed as follows:

First, the manipulator is placed in the ready position by executing **PC1**. After which, **PC2** is immediately executed placing the manipulator at a pre-programmed bearing and configuration to facilitate a safe approach to the target. **PC2** then repeatedly calls **SPC1** to obtain ranging data to the target. **PC2** utilizes the range information to determine if the vehicle is in range of the target, and if the environment is conducive to executing the control task. It does this by determining if the target is within the manipulator's workspace and if the delivery vehicle can maintain a safe operating distance to the target. For experimental purposes, **PC2** prompts the user to manually move the cart to the proper range. Practically, the prompts would be control signals to the vehicle's prime mover. If the delivery vehicle is able to keep the target within the designated range of 65 to 75 cm for 8 seconds, **PC2** terminates, and the reasoner calls **PC3**. If the vehicle is not able to maintain a safe working distance, after a designated number of attempts, **PC2** will terminate and the reasoner will call **PC8** which returns the manipulator to the nest. **PC3** executes a control algorithm that commands the manipulator to grasp the device and move it clear of the platform. If the device is not retrieved, the reasoner will call PC3 for a second attempt. If it fails the second time, **PC8** is called to return the manipulator to the nest. Once the device is successfully retrieved, the reasoner calls **PC4** to check if the environment is stable enough to place the device. If the environment is not stable, **PC7** is called to return the device to its staging area on the platform. After which **PC8** is called to return the manipulator to the nest. If the environment is stable enough, **PC5** is called to control the manipulators approach to the target. **PC5** positions the device 20 cm from the target of interest and closes the distance at approximately 1 cm/sec. While the approach is being made, **PC5** maintains the device at the commanded range even if the distance between the platform and the target is varying. If the error between the command distance and the device is greater than 2 cm the approach is abated, otherwise the approach is continued. After a designated number of attempts to approach the target are made, **PC5** will terminate and the reasoner will execute **PC7** and **PC8** respectively. When contact is made, **PC5** is terminated and the reasoner calls **PC6**. **PC6** utilizes the Zebra-ZERO's force control mode to apply a force to the device for 2.5 seconds. **PC6** terminates after the gripper releases and clears the device. The reasoner then executes **PC8**.

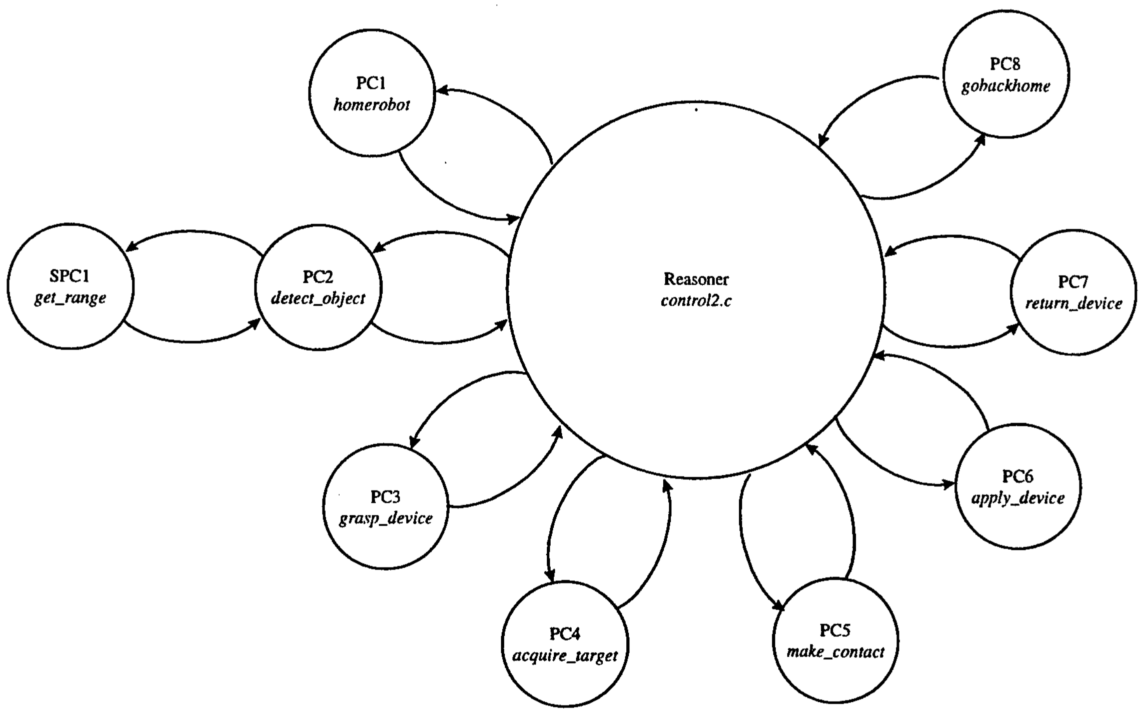

Figure 12. Program *control2.c* Controller Model

2. Results

The algorithm contained in *control2.c* performed extremely well. The controller was able to consistently place the device on the target with reasonable random perturbations where the target was kept within the manipulator's workspace. However, this simulation revealed limitations in the Zebra-ZERO's ability to respond to changes in range.

The manipulator's response varied with the distance the tool frame had to travel between movements. While the Zebra-ZERO is executing a manipulator movement function it freezes all other software functions until the movement has been completed. During this time, a relatively large range error may develop.

This limitation was analyzed to determine how much the response of the manipulator could be affected. The movement commands used to position the manipulator in *control2.c* were run at full speed using the program *mv_time.c* which is listed in Appendix A. The average time over 20 positioning commands at the same travel distance were tabulated for ranges between 0.0 and 20 cm. Figure 13 displays a graph of the experimental average time it takes to position the tool frame versus the distance to be traveled. The graph reveals that the control system could be blind to changes in position from 0.8 to 1.4 seconds for variations in range between 0.0 and 20 cm. Therefore, for the extreme case where the manipulator is advancing toward a target that is 25 cm away, a change in relative range greater than 5 cm in less than 1.4 seconds could cause damage to the manipulator or to the target.

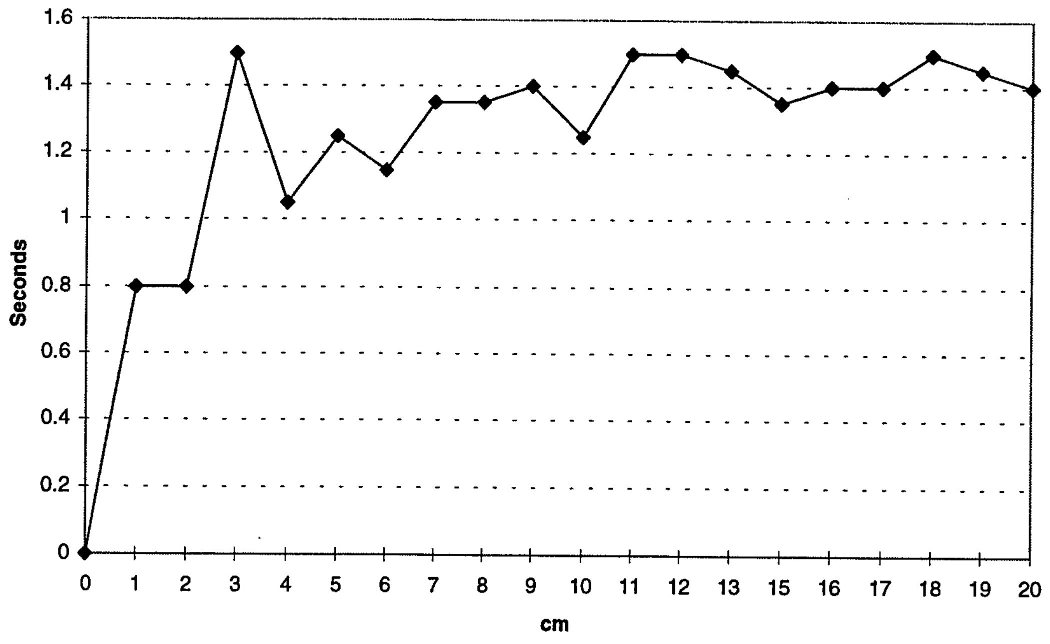

Figure 13. Zebra-ZERO Manipulator Positioning Time vs Distance.

VI. CONCLUSION AND RECOMMENDATIONS

A. THESIS SUMMARY

The capabilities of the Zebra-ZERO Force control robot were successfully expanded to include a sonar ranging system as an additional external sensor. The added sensor allowed feasibility tests to be conducted where the manipulator was mounted on a cart that simulated an autonomous vehicle. The tests showed that the Zebra-ZERO is an adequate hardware test platform on which to develop algorithms and supporting hardware that facilitate system employment in an environment where the manipulator is mounted on a moving platform.

Hardware simulations were conducted where the movements of the cart and the target object were restricted to the axial direction. The simulations showed that the Zebra-ZERO mounted on a cart that experienced random movement could operate in an environment where the target object is either stationary or in motion. The control algorithms successfully utilized the sonar system to guide the manipulator to the target utilizing position control mode of operation. After the manipulator contacted the target, the control algorithm was able to execute a seamless transition to force control mode of operation in order to manipulate the target object utilizing the force sensor.

B. FUTURE WORK

Initial hardware tests revealed that the sonar transducer may experience noise while the manipulator is in operation. During early testing some noise was experienced. However, it was so minute that it did not detrimentally affect the operation of the sonar ranging system. However, after rigorous use, the performance of the power amplifiers slowly degrades due to heat stress. It is suspected that as the power amplifiers degrade, they emit components of the PWM control signal in the form of electromagnetic energy (EMI). These emissions show up as noise imposed on the sonar transducer and its associated wiring. It is recommended that further studies include developing a solution that would attenuate not only the noise introduced by the degrading power amplifiers but also noise that may be introduced in the control environment by external sources.

Testing also revealed that although the manipulator system responded well to slow perturbations in range to the target, the system could not handle large and frequent changes in range. The manipulator responded quickly to changes that were less than 3 cm where the manipulator response time was less that one second. However, as the distance error increased it took up to 1.5 seconds for the manipulator to respond to a command. It is recommended that a new set of control commands be developed that will minimize the time overhead inherent in the commands available in the existing Zebra-ZERO software library.

The most obvious limitation of the simulations was that they only addressed movement in the axial direction. In a real-world situation movement may occur in three dimensions. It is recommended that future work include implementing sensors to resolve errors that occur in three dimensions. Additional sonar sensors may be utilized or other types of sensors such as lasers and cameras may be introduced.

In conclusion, the Zebra-ZERO proved to be a satisfactory test bed for the concepts considered in this thesis. Critical developmental issues leading to the employment of an autonomous vehicle that utilizes a manipulator to execute its mission may clearly be addressed utilizing the Zebra-ZERO Force Control Robot.

APPENDIX A. ZEBRA-ZERO DEMONSTRATION PROGRAMS

```
/*****************************************************************/
/************************* A_TEST.C *****************************/
/*****************************************************************
*
*  A_TEST.C:  a_test.c utilizes common Zebra-ZERO positioning
*  commands to demonstrate the capabilities of the manipulator.
*
*  R.A. Raphael                                            Jan 98
******************************************************************/

****************************** include files ***********************/

#include <stdio.h>
#include <stdlib.h>
#include <graphics.h>
#include <math.h>
#include <conio.h>
#include <bios.h>
#include "robot.h"

/*********************************************************************
*                          main program
**********************************************************************
/

void main(void)
{
int   delay = 10000;
vect  v3;
vect6 joint_vector;
frame tool_frame, desired_frame;
vect6 ready;
vect6 v;

/* define a joint vector 'ready' corresponding to the ready position */
mkv6(ready, 0, 90*DTOR, -180*DTOR, 180*DTOR, 0, 0);

clrscr();
homerobot();

printf("CLEAR THE AREA 2 FEET AROUND THE MANIPULATOR \n");
printf("Then press any key\n");
getch();
clrscr();
set_seg_time(5);
jog(mkv3(v3, 250, 0, -150));
sjmove(0,  170*DTOR);
sjmove(0, -170*DTOR);
sjmove(0,    0*DTOR);
clrscr();
set_seg_time(0);
```

```
/* extract the current joint angles and tool frame and save as variables
*/
where(joint_vector, &tool_frame);
where(joint_vector, &desired_frame);

/* redesignate x, y, z manipulator locations */
desired_frame.p[0] =  260;
desired_frame.p[1] =  380;
desired_frame.p[2] = -125;

printf("Use cmove to position the tool frame at base frame location [26,
38, -12.5]\n\n");
usec_timer_init();
usec_timer(delay);
/* move tool frame */
cmove(&desired_frame);

printf("Use jmove to move the manipulator through its workspace\n\n");
usec_timer_init();
usec_timer(delay);

jmove(ready);
jmove(mkv6(v,0*DTOR, 30*DTOR, -125*DTOR, 178*DTOR, 16*DTOR, 116*DTOR));
jmove(mkv6(v,-65*DTOR, -3*DTOR,  -126*DTOR, 200*DTOR, -45*DTOR,
85*DTOR));
jmove(mkv6(v,-115*DTOR,  -4*DTOR,  -90*DTOR,  230*DTOR, -45*DTOR,
115*DTOR));
jmove(mkv6(v,-153*DTOR, -10*DTOR, -118*DTOR, 184*DTOR,  -65*DTOR,
123*DTOR));
jmove(mkv6(v, 0*DTOR, 90*DTOR, -180*DTOR, 180*DTOR,  0*DTOR,   0*DTOR));
jmove(mkv6(v,60*DTOR, 17*DTOR, -170*DTOR, 154*DTOR, -31*DTOR, 95*DTOR));
jmove(mkv6(v,175*DTOR, 5*DTOR, -160*DTOR, 145*DTOR,  -60*DTOR,
85*DTOR));
jmove(ready);

printf("slightly move the joints of the manipulator using the command
jjog\n\n");
usec_timer_init();
usec_timer(delay);

set_seg_time(5);
jjog(mkv6(v,10*DTOR, 15*DTOR, -15*DTOR, 15*DTOR,  10*DTOR,   10*DTOR));
jmove(ready);
set_seg_time(0);

printf("Use hjog to move the manipulator by the displacement vector ( 5,
5, 10)\n\n ");
usec_timer_init();
usec_timer(delay);
hjog(mkv3(v3, 50.0, 50.0, 100.0));
jmove(ready);

printf("Use jog to move the tool frame relative to the base frame \n\n");
usec_timer_init();
usec_timer(delay);
jog(mkv3(v3, 40, 100, 50));
```

```
printf("reposition the manipulator using sjmove\n\n");
usec_timer_init();
usec_timer(delay);
set_seg_time(4);
sjmove(4, 40*DTOR);
sjmove(0, 40*DTOR);
sjmove(1, 100*DTOR);
set_seg_time(0);

gobackhome();
clrscr();
}  // end main


/*****************************************************************/
/************************** CONTOUR.C *****************************/
/*****************************************************************
*  CONTOUR.C:  The program contour.c utilizes force threshold sensing and
*  position control to guide a tool, such as a pencil , along a
*  contoured surface.  The program demonstrates the Zebra-ZERO's ability
*  to implement the force sensor to detect changes in the environment
*  and to adjust the tool frame according to the detected changes
*  using position control.
*
*R.A. Raphael                                                Mar 98
*******    ********************************************************/

/************************** include files ****************************/

#include <stdio.h>
#include <stdlib.h>
#include <graphics.h>
#include <math.h>
#include <conio.h>
#include <bios.h>
#include "robot.h"

void main(void)
{
int i,force_flag;
float surface_detect, force_Z, scale_factor, push_time, tolerance;
vect v3;
vect6 ready, point1, forces, v;

/* define a joint vector 'ready' corresponding to the ready position */
mkv6(ready, 0, 90*DTOR, -180*DTOR, 180*DTOR, 0, 0);
mkv6(point1, -87*DTOR, 41.3*DTOR, -192*DTOR, 180.5*DTOR, 30.4*DTOR,
12*DTOR);
scale_factor=100;

homerobot();
clrscr();
jmove(ready);          // move to ready position
gripper(40,10, 500);   // open gripper to grasp tool
printf("Place tool in gripper and press any key \n");
getch();
gripper(2,2,500);      // close gripper on tool
```

```
printf("Press any key to begin \n");
getch();
jmove(point1);          // move to point directly in front of surface
set_seg_time(5);        // slow the time required to reach point
jog(mkv3(v3, 0, 0,-50));   // close distance to surface
set_seg_time(0);        // restore segment time to max value
zero_force();           // set force sensor offset
read_user_force(forces); // read forces w/ offset
force_Z =abs(scale_factor*forces[2]); // extract Z axis value
printvect(forces);

surface_detect = 10;
tolerance = 7;
force_flag=0;
printf("Normal force:          %6.4f\n",force_Z);
printf("Surface detct force: %6.4f\n",surface_detect);

for ( i = 1; i < 200; i++)
 {
  read_user_force(forces);
  force_Z = scale_factor*forces[2];

  if (-force_Z < (surface_detect-tolerance) )
   {
    hjog(mkv3(v3, 1, 0, 1));
    force_flag = 1;
   }
  if (-force_Z > (surface_detect + tolerance) )
   {
    hjog(mkv3(v3, 1, 0, -1));
    force_flag = 1;
   }
  if(!force_flag)
  {
   jog(mkv3(v3, 1, 0, 0));
  }
  force_flag=0;
   printf("exerted force =  %6.0f\n",force_Z);
 }

jog(mkv3(v3, 0, 0, 100));   // open distance to surface
jmove(ready);
printf("Press any key when ready to release tool\n");
getch();
gripper(40,10, 500);
printf("Press any key to have the manipulator return to the nest\n");
getch();
gobackhome();
}  /* end contour */
```

```
/*****************************************************************/
/************************* FORCE.C ****************************/
/******************************************************************

/******************************************************************
*   FORCE.C:  The program force.c implements the key force
*   control functions and allows the user to experiment with the
*  various force
*  control parameters.  The algorithm first moves the arm to the
*  ready position.  Next, it sets the initial force threshold,
*  stiffness, bias, damping and push time.  It then gives the
*  user an opportunity to change the stiffness and damping vectors
*  as well as the bias force, and the amount of time the force is
*  applied.  After all parameters have been entered, push_with_bias
*  is executed.  The interactive portion of the program is
*  iterative and allows the user to keep or change parameters each
*  time it is run.  When the user is finished, the program may be
*  exited when prompted.
*
* R.A. Raphael                                                Mar98
*******************************************************************/

/**************************** include files ***********************/

#include <stdio.h>
#include <stdlib.h>
#include <graphics.h>
#include <math.h>
#include <conio.h>
#include <bios.h>
#include "robot.h"

/****************************** functions ***************************/
int near_pos(vect6 v, vect6 jv)
{
int i;

for (i=0; i<6; i++) if ( fabs(v[i]-jv[i]) > 5.0*DTOR ) return(0);
return(1);
}

void printvect_deg(vect6 v)
{
int i;

for (i=0; i<6; i++) printf("%12.3f",v[i]*RTOD);
printf("\n");
}

int read_vect6(char *param, vect6 v, int convert)
{
int i;

if (sscanf(param,"%f%f%f%f%f%f",&v[0],&v[1],&v[2],&v[3],&v[4],&v[5])!=6)
  {
  printf("Missing or invalid vect6\n");
  return(-1);
  }
if (convert)       /* convert to radians */
  for (i=0; i<6; i++) v[i] *= DTOR;
return(0);
}

int read_vect3(char *param, vect6 v)
{
if (sscanf(param,"%f%f%f",&v[0],&v[1],&v[2]) != 3)
  {
  printf("Missing or invalid vect3\n");
  return(-1);
  }
return(0);
}

int read_char(char *param, vect6 v)
{
if (sscanf(param,"%f",&v[0]) != 1)
  {
  printf("Missing or invalid vect3\n");
  return(-1);
  }
return(0);
}

char cfilename[40] = "";
char hfilename[40] = "";
FILE *cfile, *hfile;

int teach(char *varname, char mode)
{
vect6 v;
frame f;
static char spaces[] = " ";

if (!stricmp(cfilename,""))
  {
  printf("File name (w/out extension):");
  scanf("%s",cfilename);
  printf("\n");
  strcpy(hfilename,cfilename);
  strcat(cfilename,".c");
  strcat(hfilename,".h");
  }

if ( !(cfile=fopen(cfilename,"at")) )
  {
  printf("Could not open '%s'\n",cfilename);
  return(-1);
  }
if ( !(hfile=fopen(hfilename,"at")) )
  {
  fclose(cfile);
  printf("Could not open '%s'\n",hfilename);
  return(-1);
  }
```

```
where(v,&f);

if (mode == 'j')
  {
  fprintf(hfile,"extern vect6 %s;\n",varname);
  fclose(hfile);

  fprintf(cfile,"vect6 %s = { %8.4f,%8.4f,%8.4f,%8.4f,%8.4f,%8.4f };\n",
                           varname,v[0],v[1],v[2],v[3],v[4],v[5]);
  fclose(cfile);
  }
else
  {
  fprintf(hfile,"extern frame %s;\n",varname);
  fclose(hfile);

  fprintf(cfile,"frame %s = {
%9.4f,%9.4f,%9.4f,\n",varname,f.r[0][0],f.r[0][1],f.r[0][2]);
  spaces[strlen(varname)] = '\0';
  fprintf(cfile,"%s
%9.4f,%9.4f,%9.4f,\n",spaces,f.r[1][0],f.r[1][1],f.r[1][2]);
  fprintf(cfile,"%s
%9.4f,%9.4f,%9.4f,\n",spaces,f.r[2][0],f.r[2][1],f.r[2][2]);
  fprintf(cfile,"%s           %9.2f,%9.2f,%9.2f
};\n",spaces,f.p[0],f.p[1],f.p[2]);
  spaces[strlen(varname)] = ' ';

  fclose(cfile);
  }
while (_bios_keybrd(1)) getch();
return(0);
}

void plot_fig1(void)
{

printf("\n\n ");
  printf("           (-y)        \n");
  printf("            ^          \n");
  printf("            |          \n");
  printf("            |          \n");
  printf("      (x)<---x--->(-x)\n");
  printf("            |          \n");
  printf("            |          \n");
  printf("            V          \n");
  printf("           (y)         \n");
  printf("\n\n " );

} // end plot_fig1

/*******************************************************************/
```

```
/*****************************************************************/

/* Function:  force_test

   Inputs:  none

   Outputs: none

   Return value: none

   FORCE_TEST is a tool used to expose the user with the Zebra-zero's
   force control mode of operation.  The program does the following:

   Places arm in the ready position
   Rotates gripper 90 degrees
   Prompts user for specified force control parameters
   exerts specified forces
   returns to ready position
   Prompts user to run test again

   R.A. Raphael                                           Jan98
______________________________________________________________________*/

void force_test(void)
{
int i, flag;
float surface_detect, force_Z, push_time, damping_value;
vect v3;
vect6 ready, point1, forces,v, v_stiff, v_bias;
char *param, command[80], inputs[80];
/* define a joint vector 'ready' corresponding to the ready position */
mkv6(ready, 0, 90*DTOR, -180*DTOR, 180*DTOR, 0, 100*DTOR);
mkv6(point1, -70*DTOR, 20*DTOR, -180*DTOR, 181*DTOR, 21*DTOR, 30*DTOR);

clrscr();
printf("\n\n This program is used to study the functions \n\n");
printf(" set_stiffness, set_bias_force, and push_with_bias.\n");
printf("\n            --- Hit any key to proceed ---\n");
getch();

jmove(ready);          // move to ready position

// remove comment to enable the set force call
//set_force_threshold(mkv6(v, .0, .1, .1 , 500, 500, 500));

push_time = 15; // designate the number of seconds to apply force

//set default values
mkv6(v_stiff, .15, 0.15, .0, .1,.1, 0.1);
mkv6(v_bias, 0.0, 0.0, 0.2, 0, 0, 0);
damping_value = 0.35;

while (flag != 0) // Start interactive portion of function
 {
  //************** specify stiffness vector ************************

  clrscr();
  printf("\n\n  Enter the 6 element vector that specifies stiffness.
\n\n");
  printf("  The maximum value for each element is.\n");
  printvect(MAX_STIFF);
  printf("  The current stiffness values are:.\n");
  printvect(v_stiff);
  printf("\n");
  plot_fig1();

  printf("FORCE CONTROL>> ");

  gets(inputs);
  clrscr();

  if (read_vect6(inputs,v,0)== 0 )
   {
   for (i=0; i<6; i++) v_stiff[i] = v[i];
   }
  else
   {
    clrscr();
    printf("\n  No modifications received. \n");
    printf("  Using previous stiffness values \n");
    delay(10000);
   }
  set_stiffness(v_stiff);


  //************** specify bias force vector ************************

  clrscr();
  printf("\n\n  Enter the 6 element vector that specifies bias force.
\n\n");
  printf("  The maximum value for each element is \n");
  printvect(MAX_BIAS);
  printf("  The current stiffness values are: \n");
  printvect(v_bias);
  printf("\n");
  plot_fig1();
  printf("FORCE CONTROL>> ");

  gets(inputs);
  clrscr();

  if (read_vect6(inputs,v,0)== 0 )
   {
   for (i=0; i<6; i++) v_bias[i] = v[i];
   }
  else
   {
    clrscr();
    printf("\n  No modifications received.  \n");
    printf("  Using previous bias values \n");
    delay(10000);
   }

  set_bias_force(v_bias);


  //************** specify damping value ************************
  clrscr();
  printf("\n\n  Enter a value between 0.0 and 0.35 to specify damping
constant. \n\n");
  printf(" set_damping sets a damping constant for all force controlled
motions \n");
  printf(" Lower values correspond to more damping.  If motions appear
unstable, \n");
  printf(" the damping value should be lowered; if the force response is
too \n");
  printf(" sluggish, the damping value should be increased.");
  printf("  The current damping value is: %1.2f\n",damping_value);
  printf("\n");
  printf("FORCE CONTROL>> ");

  gets(inputs);
  clrscr();

  if (read_char(inputs,v)== 0 )
   {
    damping_value = v[0];
   }
  else
   {
    clrscr();
    printf("\n  No modification received.\n");
    printf("  Using previous damping value\n");
    delay(10000);
   }

  set_damping(damping_value);

  /************************ Specify Push time **********************/

  clrscr();
  printf("\nEnter the duration in seconds the force is to be applied.
\n\n");
  printf("FORCE CONTROL>> ");
  gets(inputs);
  clrscr();

  if (read_char(inputs,v)== 0 )
   {
    push_time = v[0];
   }
  else
   {
    clrscr();
    printf("\n  No modification received.\n");
    printf("  Using previous duration value\n");
    delay(10000);
    }

   set_damping(damping_value);

   // ************** execute PUSH WITH BIAS ******************
   clrscr();
   printf("\nThe stiffness vector is \n");
   printvect(v_stiff);
   printf("\nThe bias vector is \n");
   printvect(v_bias);
   printf("\nThe damping constant is: %1.2f\n",damping_value);
   printf("\nThe duration that push_with_bias is to be applied is
%2.2f\n",push_time);
   plot_fig1();
   printf("\n Press any key to execute push_with_bias");
   gets(inputs);

   zero_force();

   push_with_bias(push_time);
   stiffness_off();

   jmove(ready);

   printf("  would you like to continue?");

   printf("  Press any key if yes, enter n if no \n\n");
   printf("FORCE CONTROL>> ");
   gets(command);
    if (!stricmp(command,"n")) flag = 0;
    else flag = 1;

  }

stiffness_off();

}  /* end test_force */


/*************************************************************************
*/
/*                           main program
*/
/*************************************************************************
*/

void main(void)
{
clrscr();
homerobot();
force_test();
gobackhome();

}
```

```
/*****************************************************************/
/************************** RANGER.C ******************************/
/******************************************************************

/*ULDI01.C*********************************************************

File:                          RANGER.C

Library Call Demonstrated:     ranger()

Purpose:                       Reads a digital input port.

Demonstration:                 Configures FIRSTPORTA for input and
                         reads the value on the port,
                         then displays the sonar ranging
                         information in cm.


Other Library Calls:           cbDConfigPort()
                         cbErrHandling()

Special Requirements:          Board 0 must have a digital input port.


Moodified 12/2/98 by R.A. Raphael for use with the Polaroid sonar ranger
This program is derived form ULDIO1.C

ULDIO1.C has the following Copyright:
(c) Copyright 1995, ComputerBoards, Inc.
All rights reserved.

modified 12/2/98
*************************************************************************
*/

/* Include files */
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <math.h>
#include "cb.h"

/* Prototypes */
void ClearScreen (void);
void GetCursor (int *x, int *y);
void MoveCursor (int x, int y);

void main ()
    {
    /* Variable Declarations */
    int Row, Col, I;
    int BoardNum = 0;
    int ULStat = 0;
    int PortNum, Direction;
    int PowerVal, BitValue;
    int Zero = 0;
    int One = 1;
```

```
  unsigned DataValue;
  float    range, RevLevel = (float)CURRENTREVNUM;

/* Declare UL Revision Level */
 ULStat = cbDeclareRevision(&RevLevel);

  /* Initiate error handling
    Parameters:
        PRINTALL :all warnings and errors encountered will be printed
        STOPALL  :if any error is encountered, the program will stop */
  ULStat = cbErrHandling (PRINTALL, STOPALL);

  /* set up the display screen */
  ClearScreen();
  printf ("Demonstration of ranger()\n\n");
  printf ("Press any key to quit.\n\n");
  printf ("The first 7 bits are: ");
  printf ("0  1  2  3  4  5  6  7 \n");
  GetCursor (&Col, &Row);

  /* configure FIRSTPORTA  for digital input
    Parameters:
        BoardNum    :the number used by CB.CFG to describe this board.
        PortNum     :the input port
        Direction   :sets the port for input or output */
  PortNum = FIRSTPORTA;
  Direction = DIGITALIN;
  ULStat = cbDConfigPort (BoardNum, PortNum, Direction);

  while (!kbhit())
    {
    /* Read the 7 bits digital input and display
       Parameters:
          BoardNum    :the number used by CB.CFG to describe this board
          PortNum     :the input port
          DataValue   :the value read from the port    */
    ULStat = cbDIn(BoardNum, PortNum, &DataValue);
    DataValue = ~DataValue & 0377;
    if (DataValue < 91)
      range = 00.0;
    else
     range = DataValue*0.472;

    /* display the value collected from the port */
    MoveCursor (Col, Row);
    printf ("Range: %2.1f cm ", range);

    /* parse DataValue into bit values to indicate on/off status */
    MoveCursor (Col + 21, Row);
    for (I = 0; I < 8; I++)
        {
        BitValue = One;
        PowerVal = (int)pow(2, I);
        if (DataValue & PowerVal)
           {
           BitValue = Zero;
           }
```

```
            printf (" %u ", BitValue);
            }
        }

    MoveCursor (1, 20);
    printf ("\n");

    }

/**********************************************************************
***
*
* Name:      ClearScreen
* Arguments: ---
* Returns:   ---
*
* Clears the screen.
*
**********************************************************************
**/

#define BIOS_VIDEO   0x10

void
ClearScreen (void)
{
    union REGS InRegs,OutRegs;

    InRegs.h.ah = 0;
    InRegs.h.al = 2;
    int86 (BIOS_VIDEO, &InRegs, &OutRegs);
    return;
}

/**********************************************************************
*
*
* Name:      MoveCursor
* Arguments: x,y - screen coordinates of new cursor position
* Returns:   ---
*
* Positions the cursor on screen.
*
**********************************************************************
*/

void
MoveCursor (int x, int y)
{
    union REGS InRegs,OutRegs;
    InRegs.h.ah = 2;
    InRegs.h.dl = (char) x;
    InRegs.h.dh = (char) y;
    InRegs.h.bh = 0;
    int86 (BIOS_VIDEO, &InRegs, &OutRegs);
    return;
}
```

```
/***********************************************************************
*
*
* Name:      GetCursor
* Arguments: x,y - screen coordinates of new cursor position
* Returns:   *x and *y
*
* Returns the current (text) cursor position.
*
************************************************************************
**/

void
GetCursor (int *x, int *y)
{
    union REGS InRegs,OutRegs;

    InRegs.h.ah = 3;
    InRegs.h.bh = 0;
    int86 (BIOS_VIDEO, &InRegs, &OutRegs);
    *x = OutRegs.h.dl;
    *y = OutRegs.h.dh;
    return;
}

/*ULDI01.C*************************************************************

File:                         RANGER.C

Library Call Demonstrated:    ranger()

Purpose:                      Reads a digital input port.

Demonstration:                Configures FIRSTPORTA for input and
                         reads the value on the port,
                         then displays the sonar ranging
                         information in cm.


Other Library Calls:          cbDConfigPort()
                         cbErrHandling()

Special Requirements:         Board 0 must have a digital input port.


Moodified 12/2/98 by R.A. Raphael for use with the Polaroid sonar ranger
This program is derived form ULDIO1.C

ULDIO1.C has the following Copyright:
(c) Copyright 1995, ComputerBoards, Inc.
All rights reserved.

modified 12/2/98
```

```
************************************************************************
/

/* Include files */
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <math.h>
#include "cb.h"


/* Prototypes */
void ClearScreen (void);
void GetCursor (int *x, int *y);
void MoveCursor (int x, int y);

void main ()
    {
    /* Variable Declarations */
    int Row, Col, I;
    int BoardNum = 0;
    int ULStat = 0;
    int PortNum, Direction;
    int PowerVal, BitValue;
    int Zero = 0;
    int One = 1;
    unsigned DataValue;
    float    range, RevLevel = (float)CURRENTREVNUM;

  /* Declare UL Revision Level */
   ULStat = cbDeclareRevision(&RevLevel);

    /* Initiate error handling
      Parameters:
          PRINTALL :all warnings and errors encountered will be printed
          STOPALL  :if any error is encountered, the program will stop */
    ULStat = cbErrHandling (PRINTALL, STOPALL);

    /* set up the display screen */
    ClearScreen();
    printf ("Demonstration of ranger()\n\n");
    printf ("Press any key to quit.\n\n");
    printf ("The first 7 bits are: ");
    printf ("0  1  2  3  4  5  6  7 \n");
    GetCursor (&Col, &Row);

    /* configure FIRSTPORTA  for digital input
      Parameters:
          BoardNum    :the number used by CB.CFG to describe this board.
          PortNum     :the input port
          Direction   :sets the port for input or output */
    PortNum = FIRSTPORTA;
    Direction = DIGITALIN;
    ULStat = cbDConfigPort (BoardNum, PortNum, Direction);

    while (!kbhit())
      {
        /* Read the 7 bits digital input and display
           Parameters:
              BoardNum    :the number used by CB.CFG to describe this board
              PortNum     :the input port
              DataValue   :the value read from the port    */
        ULStat = cbDIn(BoardNum, PortNum, &DataValue);
        DataValue = ~DataValue & 0377;
        if (DataValue < 91)
          range = 00.0;
        else
         range = DataValue*0.472;

        /* display the value collected from the port */
        MoveCursor (Col, Row);
        printf ("Range: %2.1f cm ", range);

        /* parse DataValue into bit values to indicate on/off status */
        MoveCursor (Col + 21, Row);
        for (I = 0; I < 8; I++)
            {
            BitValue = One;
            PowerVal = (int)pow(2, I);
            if (DataValue & PowerVal)
              {
              BitValue = Zero;
              }
            printf (" %u ", BitValue);
            }
        }

    MoveCursor (1, 20);
    printf ("\n");

    }
/************************************************************************
**
* Name:      ClearScreen
* Arguments: ---
* Returns:   ---
*
* Clears the screen.
*
************************************************************************
*/

#define BIOS_VIDEO   0x10

void
ClearScreen (void)
{
    union REGS InRegs,OutRegs;

    InRegs.h.ah = 0;
    InRegs.h.al = 2;
    int86 (BIOS_VIDEO, &InRegs, &OutRegs);
    return;
```

```
}


/***********************************************************************
*
* Name:       MoveCursor
* Arguments: x,y - screen coordinates of new cursor position
* Returns:    ---
*
* Positions the cursor on screen.
*
***********************************************************************/

void
MoveCursor (int x, int y)
{
    union REGS InRegs,OutRegs;

    InRegs.h.ah = 2;
    InRegs.h.dl = (char) x;
    InRegs.h.dh = (char) y;
    InRegs.h.bh = 0;
    int86 (BIOS_VIDEO, &InRegs, &OutRegs);
    return;
}

/***********************************************************************
*
*
* Name:       GetCursor
* Arguments: x,y - screen coordinates of new cursor position
* Returns:    *x and *y
*
* Returns the current (text) cursor position.
*
************************************************************************
*/

void
GetCursor (int *x, int *y)
{
    union REGS InRegs,OutRegs;

    InRegs.h.ah = 3;
    InRegs.h.bh = 0;
    int86 (BIOS_VIDEO, &InRegs, &OutRegs);
    *x = OutRegs.h.dl;
    *y = OutRegs.h.dh;
    return;
}
```

```
/*****************************************************************/
/************************** MV_TIME.C *****************************/
/*****************************************************************

/***********************************************************
*
* MV_TIME: The program mv_time is used to calculate the
* average amount of time it takes to move the manipulator a
* given distance. The program utilizes the Zebra-ZERO
* movement function hjog. However, the movement function is
* interchangeable.  The function moves the tool frame
* back and forth for a designated number of cycles.
* The user is prompted to start the movement cycle.
* at the same time, a stop watch is used to time the events.
* The total time divided over the number of movements is
* the average movement time.
*
*  R.A. Raphael                                 Mar 98
***********************************************************/

/* Include files */
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <math.h>
#include <stdlib.h>
#include <graphics.h>
#include <bios.h>
#include "robot.h"
#include "cb.h"
#include <time.h>

void main(void)
{
int   i = 0;
float move_distance=200;
vect v3;
vect6  joint_vector;
frame start_frame;
get_init_data("DEFAULT.INI");

homerobot();

clrscr();
hjog(mkv3(v3, 0, 0, -200));
where(joint_vector, &start_frame);
printf("Prepare the stopwatch for timing the movements \n");
printf("Press any key to begin the movements \n");
getch();

while ( i++ < 10)
{
 hjog(mkv3(v3, 0, 0, move_distance));
 hjog(mkv3(v3, 0, 0, -move_distance));
 printf(" %d ",i);
}
printf("press any key to continue");
getch();
gobackhome();
} // end main
```

APPENDIX B. ZEBRA-ZERO HARDWARE TEST PROGRAMS FOR A STATIONARY TARGET

```
/****************************************************************/
/************************* CONTROL1.C ****************************/
/***************************************************************

/*********************************************************************
*
*
*  CONTROL1.c tests the ability of the Zebra-ZERO to perform a complex
* pick-and-place task where the manipulator is on a mobile delivery
* platform and the target object is stationary.
*
*  R.A. Raphael                                            Mar 98
*******************************************************************/

/* Include files */
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <math.h>
#include <stdlib.h>
#include <graphics.h>
#include <bios.h>
#include "robot.h"
#include "cb.h"

void main(void)
{

vect6 ready,search_config, contact_config;
int bearing=90;
int status;
int grasp_count;
int acquire_obj;
get_init_data("DEFAULT.INI");

homerobot();

status = 1;
acquire_obj = 0;            //
while(status != 0)
 {
  switch(status)
   {
    case 1:           //ensures delivery platform is properly placed
      status=acquire_object(bearing);
      if (status)
      status = 2;
      else
      status = 1;
      acquire_obj++;
```

```
        if(acquire_obj > 2)
        status = 0;
        break;

      case 2:
        status = grasp_object();;
        if (status==2)
        status = status = 1;
        else
        status = 0;
        printf("peg was not retrieved.");
        break;

     default:
       clrscr();
       printf(" An unexpected error has occured. \n");
       printf(" Placing manipulator in nest.");
       getch();
       break;

     } // end switch

   }   // end while

 gobackhome();

 }
```

```
/******************************************************************/
/************************** PHASE1.C ****************************/
/*****************************************************************

/**********************************************************************
*
*  PHASE1.C:  Used with control1.c to control the Zebra-ZERO Manipulator
on
*  a moving platform
*
*  R.A. Raphael
Mar 98
**********************************************************************
/

/* Include files */
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <math.h>
#include <stdlib.h>
#include <graphics.h>
#include <bios.h>
#include "robot.h"
#include "cb.h"

//        functions for I/O board

/**********************************************************************
*
*
* Name:      ClearScreen
* Arguments: ---
* Returns:   ---
*
* Clears the screen.
*
*********************************************************************/

#define BIOS_VIDEO   0x10

void ClearScreen (void)
{
    union REGS InRegs,OutRegs;
    InRegs.h.ah = 0;
    InRegs.h.al = 2;
    int86 (BIOS_VIDEO, &InRegs, &OutRegs);
    return;
}


/********************************************************************
*
* Name:      MoveCursor
* Arguments: x,y - screen coordinates of new cursor position
* Returns:   ---
```

```
*
* Positions the cursor on screen.
*
**********************************************************************/


void MoveCursor (int x, int y)
{
    union REGS InRegs,OutRegs;

    InRegs.h.ah = 2;
    InRegs.h.dl = (char) x;
    InRegs.h.dh = (char) y;
    InRegs.h.bh = 0;
    int86 (BIOS_VIDEO, &InRegs, &OutRegs);
    return;
}

/**********************************************************************
*
* Name:      GetCursor
* Arguments: x,y - screen coordinates of new cursor position
* Returns:   *x and *y
*
* Returns the current (text) cursor position.
*
**********************************************************************/

void GetCursor (int *x, int *y)
{
    union REGS InRegs,OutRegs;
    InRegs.h.ah = 3;
    InRegs.h.bh = 0;
    int86 (BIOS_VIDEO, &InRegs, &OutRegs);
    *x = OutRegs.h.dl;
    *y = OutRegs.h.dh;
    return;
}


int near_pos(vect6 v, vect6 jv)
{
int i;

for (i=0; i<6; i++) if ( fabs(v[i]-jv[i]) > 5.0*DTOR ) return(0);
return(1);
}

void printvect_deg(vect6 v)
{
int i;

for (i=0; i<6; i++) printf("%12.3f",v[i]*RTOD);
printf("\n");
}
```

```
/**********************************************************************
*
* Name:      acquire_object
* Arguments: bearing - bearing of closest object to manipulator
* Returns:   1 if object is found
*          0 if no objcect is found
*
*
**********************************************************************/

int acquire_object(int bearing)
{
vect  f_new, v3;
vect6 home_jv;
vect6 ready,search_config, contact_config;
vect6 v;
frame f;
char c;
int Row, Col, I, i, n;
int BoardNum = 0;
int ULStat = 0;
int PortNum, Direction;
int PowerVal, BitValue;
int False = 0;
int True = 1;
int aquire_peg, accuracy_count = 0;
int error_int;
vect6 joint_vector, grasp_config;
frame tool_frame;
frame desired_frame;
frame grasp_frame;
unsigned DataValue, DataValue_temp;
float  range,range_in, error, RevLevel = (float)CURRENTREVNUM;
float  sum_value, last_sum_value=42;
float  x_posit, y_posit,z_posit; //location of tool frame
float  Tool_Location, Tool_Location_x, Tool_Location_y; // range of tool
from base frame origion
float  desired_frame_vector;

/* Declare UL Revision Level. This is required for CYDIO I/O board. */
ULStat = cbDeclareRevision(&RevLevel);

mkv6(ready, 0.0, 90.0*DTOR, -180.0*DTOR, 180.0*DTOR, 0.0, 0.0);
aquire_peg  = False;

printf ("In LOOP bearing is %d \n  ",bearing);
mkv6(search_config, bearing*DTOR, 90.0*DTOR, -132.0*DTOR, 183.0*DTOR,
48.0*DTOR, 100.0*DTOR);
mkv6(contact_config, bearing*DTOR, 50.0*DTOR, -160.0*DTOR, 175.0*DTOR, -
17.3*DTOR, 10.2*DTOR);
jmove(search_config);
where(joint_vector, &tool_frame);

    /* Initiate error handling
      Parameters:
```

```
      PRINTALL :all warnings and errors encountered will be printed
      STOPALL  :if any error is encountered, the program will stop */
ULStat = cbErrHandling (PRINTALL, STOPALL);

GetCursor (&Col, &Row);

/* configure FIRSTPORTA  for digital input
  Parameters:
      BoardNum    :the number used by CB.CFG to describe this board.
      PortNum     :the input port
      Direction   :sets the port for input or output */
PortNum = FIRSTPORTA;
Direction = DIGITALIN;
ULStat = cbDConfigPort (BoardNum, PortNum, Direction);
ULStat = cbDIn(BoardNum, PortNum, &DataValue);
DataValue = ~DataValue & 0377;
DataValue_temp = DataValue * 0.5;
z_posit = 300;
where(joint_vector, &desired_frame);
last_sum_value=40;
where(joint_vector, &desired_frame);
accuracy_count = 0;
aquire_peg = False;
  n = 120;
  while (--n > 1)

  {
  /* Read the 7 bits digital input and display
     Parameters:
        BoardNum    :the number used by CB.CFG to describe this board
        PortNum     :the input port
        DataValue   :the value read from the port   */

  ULStat = cbDIn(BoardNum, PortNum, &DataValue);
  DataValue = ~DataValue & 0377;
  if(DataValue < 91)
   {
    usec_timer_init();
    usec_timer(1000);
    ULStat = cbDIn(BoardNum, PortNum, &DataValue);
    DataValue = ~DataValue & 0377;
    if (DataValue< 91)
     {
     range_in = 91* 0.472;
     }
    else
     {
     range_in = DataValue*0.472;
     }
   }

  else
   {
    range_in = DataValue*0.472;
   }

  // Impliment filter
```

```
        sum_value = range_in + 0.8 * last_sum_value;
        range = 0.2 * sum_value;
        last_sum_value = sum_value;
        where(joint_vector, &tool_frame);
        Tool_Location_x =  tool_frame.p[0];
        Tool_Location_y =  tool_frame.p[1];
        Tool_Location = 0.1 * sqrt( Tool_Location_x *Tool_Location_x +
Tool_Location_y * Tool_Location_y)-6;

        error = range - Tool_Location;
        desired_frame_vector = 10.0 * (error + Tool_Location + 6.0);
        desired_frame.p[0] =  desired_frame_vector * cos(bearing*DTOR);
        desired_frame.p[1] =  desired_frame_vector * sin(bearing*DTOR);
        desired_frame.p[2] =  z_posit;

        /* display the value collected from the port */
        MoveCursor (Col, Row);

        printf ("range %2.1f \n  tool location %4.1f \n error %4.1f ",
range, Tool_Location, error);

        cmove(&desired_frame);
        if (aquire_peg == False)
          {
          if ( abs((int)error) < 4)
            {
            z_posit = 150;
            }
          }
        error_int = 10 * error;

        if ( abs(error_int) < 3)
          {
           accuracy_count = accuracy_count + 1;
           printf("a count %d",accuracy_count);
          }
        else
          {
           accuracy_count = 0;
          }

        if (accuracy_count > 40)
          {

           mkv6(grasp_config, bearing*DTOR, 50.0*DTOR, -160.0*DTOR,
175.0*DTOR, -20.0*DTOR, 10.0*DTOR);
           jmove(grasp_config);
           where(joint_vector, &grasp_frame);

           grasp_frame.p[0] =  desired_frame.p[0];
           grasp_frame.p[1] =  desired_frame.p[1];
           grasp_frame.p[2] =  0.0;
           cmove(&grasp_frame);
           return(1);
          }
        }
return(0);
```

```
}

/**********************************************************************
*
* Name:      contact_object
* Arguments: None
* Returns:   1 if surface is contacted
*          0 if the surface is not detected
*
* Places the gripper so that it is contacting the peg.
*
* R.A. Raphael                                                Mar 98
*********************************************************************/

int contact_object()
{
int n;
vect6 force_vector;
vect v3;
float force_z, displacement;

zero_force();
n = 40;
while (--n > 0)
 {
  read_user_force(force_vector);
  force_z = abs(100*force_vector[2]);// extract z force
  if (force_z > 10)
   {
    printf("\n surface detected\n \n");
    return(1);
   }
  hjog(mkv3(v3, 0, 0, 1));

 } // end while
hjog(mkv3(v3,0,0,-20));
return(0);
} // end function


/**********************************************************************
*
*
* Name:      extract_peg
* Arguments: None
* Returns:   None
*
* extracts peg.
*
**********************************************************************
/

int extract_peg(void)
{
vect v3;
vect6 force_vector;
float force_x;
```

```
set_seg_time(5);
hjog(mkv3(v3,0, 0, -5));
gripper(80, 5, 0);
hjog(mkv3(v3, 0, 0, 25));
//getch();
zero_force();
gripper(0, 10, 1000);
jog(mkv3(v3, 0, 0, 30));
jog(mkv3(v3, 0, 0, 30));

read_user_force(force_vector);
  force_x = abs(100*force_vector[0]);// extract z force
  if (force_x > 5)
   {
    printf("\n peg retreaved detected\n \n");
    return(1);
   }
return(0);
} // end function

/***********************************************************************
*
*
* Name:      place_peg
* Arguments: None
* Returns:   None
*
* places peg in front of hole.
*
***********************************************************************
/

int place_peg(void)
{
vect v3;
vect6 force_vector;
int n;
float force_x;
set_seg_time(5);
hjog(mkv3(v3, 0, 0, -80));
jog(mkv3(v3, 0, 0, -100));
zero_force();
set_seg_time(1);
n = 100;
while(--n > 0)
 {
  read_user_force(force_vector);
  force_x = abs(100*force_vector[0]);// extract y force
  if (force_x > 10)
   {
    printf("\n peg placed\n \n");
    gripper(80, 5, 0);
    set_seg_time(0);
    return(1);
   }
  jog(mkv3(v3, 0, 0, -1));
 }
```

```
printf(" peg not placed");
set_seg_time(0);
return(0);
} // end function
/**************************************************************************
*
*
* Name:      grasp_object
* Arguments: bearing - bearing of closest object to manipulator
* Returns:   1 if object is found
*           0 if no objcect is found
*
*
***********************************************************************/
int grasp_object(int bearing)
{
vect  f_new, v3;
vect6 home_jv;
vect6 ready,search_config, contact_config;
vect6 v;
frame f;
char c;
int    status;
vect6 joint_vector, grasp_config;
frame tool_frame;
frame desired_frame;
frame grasp_frame;

set_seg_time(5);              // slow manipulator response
hjog(mkv3(v3,-119, 0, 0));    // postion gripper infront of peg
printf("\n grasp object");
set_seg_time(0);
status = contact_object();
if (status < 1)
  return(2);

extract_peg();
if (status < 1)
  return(0);

place_peg();
if (status < 1)
  return(0);

jog(mkv3(v3, 0, 0, 200));
set_seg_time(0);
return(1);
}
```

APPENDIX C. ZEBRA-ZERO HARDWARE TEST PROGRAMS FOR A MOVING TARGET

```
/*******************************************************************/
/************************** CONTROL2.C *****************************/
/*******************************************************************

/**********************************************************************
*
*  CONTROL2.C:  executes a control algorithm that grasps a
*  device whose location and configuration is *known by the control
*  algorithm, and places it on a vertical surface where the *distance
*  between the *manipulator and the surface may be constantly varying.
*  Must be compiled with force.c, robot.lib and cbcl.lib.
*
*  R.A. Raphael                                              Mar 98
*********************************************************************/

/* Include files */
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <math.h>
#include <stdlib.h>
#include <graphics.h>
#include <bios.h>
#include "robot.h"
#include "cb.h"
#include <time.h>

void main(void)
{

vect6 ready,search_config, contact_config;
int bearing=90;
int status;
int grasp_count;

get_init_data("DEFAULT.INI");

homerobot();

status = 1;
grasp_count = 0;            //
while(status != 0)
 {
  switch(status)
   {
    case 1:          //ensures delivery platform is properly placed
      status=detect_object(bearing);
      if (status)
      status = 2;
```

```
        break;
      case 2:
        status = grasp_device();
        if (status)
        status = 3;
        else
        status = 2;
        grasp_count++;

        if(grasp_count>2)
        status = 0;
        break;

      case 3:
        status = acquire_target(bearing);
        if(status)
        status = 4;
        else
        status = 6;
        break;

    case 4:
      status = make_contact(bearing);
      if (status)
        status = 5;
      else
        status = 6;
      break;

    case 5:
      apply_device();
      status = 0;
      break;

    case 6:
      return_device();
      status = 0;
      break;

    default:
      clrscr();
      printf(" An unexpected error has occured. \n");
      printf(" Placing manipulator in nest.");
      getch();
      break;

    } // end switch

  }   // end while

gobackhome();

clrscr();

} // end main
```

```
/*******************************************************************/
/************************** PHASE2.C *****************************/
/*******************************************************************

/********************************************************************
*  PHASE2.C:  Used with control2.c to control the Zebra-ZERO
*  Manipulator on a moving platform.
*
*  R.A. Raphael                                              Mar 98
*********************************************************************/

/* Include files */
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <math.h>
#include <stdlib.h>
#include <graphics.h>
#include <bios.h>
#include "robot.h"
#include "cb.h"
#define  number_of_scans   60
#include <time.h>

//        functions for I/O board
/*********************************************************************
*
*
* Name:      ClearScreen
* Arguments: ---
* Returns:   ---
*
* Clears the screen.
*
*
*********************************************************************/

#define BIOS_VIDEO   0x10

void ClearScreen (void)
{
    union REGS InRegs,OutRegs;
    InRegs.h.ah = 0;
    InRegs.h.al = 2;
    int86 (BIOS_VIDEO, &InRegs, &OutRegs);
    return;
}

/*********************************************************************
*
*
* Name:      MoveCursor
* Arguments: x,y - screen coordinates of new cursor position
* Returns:   ---
*
* Positions the cursor on screen.
*
```

```
**********************************************************************
/

void MoveCursor (int x, int y)
{
    union REGS InRegs,OutRegs;

    InRegs.h.ah = 2;
    InRegs.h.dl = (char) x;
    InRegs.h.dh = (char) y;
    InRegs.h.bh = 0;
    int86 (BIOS_VIDEO, &InRegs, &OutRegs);
    return;
}

/*********************************************************************
***
*
* Name:      GetCursor
* Arguments: x,y - screen coordinates of new cursor position
* Returns:   *x and *y
*
* Returns the current (text) cursor position.
*
**********************************************************************
**/

void GetCursor (int *x, int *y)
{
    union REGS InRegs,OutRegs;

    InRegs.h.ah = 3;
    InRegs.h.bh = 0;
    int86 (BIOS_VIDEO, &InRegs, &OutRegs);
    *x = OutRegs.h.dl;
    *y = OutRegs.h.dh;
    return;
}

int near_pos(vect6 v, vect6 jv)
{
int i;

for (i=0; i<6; i++) if ( fabs(v[i]-jv[i]) > 5.0*DTOR ) return(0);
return(1);
}

void printvect_deg(vect6 v)
{
int i;

for (i=0; i<6; i++) printf("%12.3f",v[i]*RTOD);
printf("\n");
}
```

```
/***********************************************************************
*
* Name:      get_range
* Arguments: None
* Returns:   Range in cm
*
* get_range polls the I/O board input port to acquire the average sonar
* ranging data over 5 readings taken over a time designated as "delay."
* It then determines if the reading is within range.  Max range reading
* is 123, min range reading is 42 cm. The highest value this function
* will return is 123 and the lowest is 41.  Numbers that are detected as
* being less than 41 are returned as 200.
*
*
* R.A. Raphael Mar 1998
***********************************************************************/

int get_range(void)
{

/* Variable Declarations */
    int Row, Col, I, i;
    int BoardNum = 0;
    int ULStat = 0;
    int PortNum, Direction;
    int PowerVal, BitValue;
    int acquire, contact;
    int delay = 1500;

    unsigned DataValue; // DataValue_temp;
    float  range;
    float  range_in, error, RevLevel = (float)CURRENTREVNUM;
    float  sum_value, last_sum_value=42;
    float  bearing; // bearing to object
    float  x_posit, y_posit; //location of tool frame
    float  Tool_Location; // range of tool from base frame origin

/* Declare UL Revision Level. This is required for CYDIO I/O board. */
ULStat = cbDeclareRevision(&RevLevel);


    /* Initiate error handling
      Parameters:
          PRINTALL :all warnings and errors encountered will be printed
          STOPALL  :if any error is encountered, the program will stop */
    ULStat = cbErrHandling (PRINTALL, STOPALL);


    /* configure FIRSTPORTA  for digital input
      Parameters:
          BoardNum    :the number used by CB.CFG to describe this board.
          PortNum     :the input port
          Direction   :sets the port for input or output */
    PortNum = FIRSTPORTA;
    Direction = DIGITALIN;
    ULStat = cbDConfigPort (BoardNum, PortNum, Direction);
    ULStat = cbDIn(BoardNum, PortNum, &DataValue);
    DataValue = ~DataValue & 0377;


      // obtain the average value of 5 consecutive sonar readings
      for ( i = 0; i < 5; ++i)
      {
      /* Read the 7 bits digital input and display
         Parameters:
            BoardNum    :the number used by CB.CFG to describe this board
            PortNum     :the input port
            DataValue   :the value read from the port    */

      ULStat = cbDIn(BoardNum, PortNum, &DataValue);
      DataValue = ~DataValue & 0377;


      if (DataValue < 91)    // screen for minimum reading
        range_in = 90*0.472;// scale sonar reading to cm
      else
        range_in = DataValue*0.472;// scale sonar reading to cm

      // execute averaging over 5 samples
      sum_value = range_in + last_sum_value;
      range = 0.2 * sum_value;
      last_sum_value = sum_value;

      usec_timer_init(); // initialize timer
      usec_timer(delay); // start timer
      }

      if ( range < 40)    // set min range flag of 200
        range = 200;

return(range);
}

/**************************************************************************
*
* Name:       detect_objec
* Arguments: bearing
* Returns:    1 if object is found
*           0 if no object is found
*
* detect_object places the manipulator in "detect configuration." This
* configuration moves the manipulator limbs out of the way of the
* transducer.  The function "get_range" is called to obtain the range to
* the target. Target must be between min_range and max_range for a min of
* n_scans calls to "get_range" for a 1 to be returned.  n scans are
* designated to get 8 consecutive valid ranges.  After which, a 0 is
* returned.
*
* R.A. Raphael Mar 1998
*************************************************************************/

int detect_object(int bearing)
{
```

```
int    range;
int    n = 100;
int    detect_valid = 0;
int    min_range = 65;
int    max_range = 75;
int    n_scans=8;
int    Col, Row;
vect6 detect_config;

// place  manipulator sonar sensor at bearing to detect object
mkv6(detect_config, bearing*DTOR, 100.0*DTOR, -175.0*DTOR, 185.0*DTOR,
18.0*DTOR, 0.0*DTOR);
jmove(detect_config);

// track approach to object of interest

clrscr();
MoveCursor (1, 10);
GetCursor (&Col, &Row); // memorize cursor location

while ( --n > 0 )
 {
  range=get_range();    // fetch range
  MoveCursor (Col, Row);// place cursor

  /*********** prompt user to position manipulator ****************/

  if (range > 150 || range < 60 )
   {
    printf("Target is too close. Move manipulator away form target \n");
    printf("                                                       \n");
    printf("                                                       \n");
   }

  else if (range < 100)
   {
    printf("Target is in range and must remain between 65 & 75cm  \n");
    printf("away form manipulator for 8 seconds.                   \n");
    printf("Range to detected object is %d                         \n",
range);
   }
  else
   {
    printf("Target out of range approach Target.                   \n");
    printf("                                                       \n");
    printf("                                                       \n");
   }

  // if range is within min and max range for n_scans return 1
  if(range < max_range && range > min_range)
    detect_valid = detect_valid + 1;
  else
    detect_valid = 0;

  if(detect_valid > n_scans)
    return(1);
 } // end while in_range loop

return(0); // return 0 if could not detect object

} // end function detect_objec

/***************************************************************************
*
* Name:     grasp_device
* Arguments: none
* Returns:   1 if device is retrieved
*          0 if object is not retrieved
*
*   grasp_device retrieves a known device from a known locaion a
positions
* the manipulator with device in hand for the next task.
*
* R.A. Raphael  Mar 1998
**************************************************************************/

int grasp_device(void)
{
vect v3;
vect6 force_vector;
float force_z;
vect6 grasp_vector, contact_config;

clrscr();
MoveCursor (1, 10);
mkv6(grasp_vector, 15.0*DTOR, 48.6*DTOR, -174.3*DTOR, 182.2*DTOR,
59.0*DTOR, 20.0*DTOR);
mkv6(contact_config, 0.0*DTOR, 90.0*DTOR, -132.0*DTOR, 183.0*DTOR,
48.0*DTOR, 100.0*DTOR);

jmove(grasp_vector);        // Maneuver manipulator over device to be
placed
gripper(80.0, 10, 0);       // open gripper
set_seg_time(4);            // slow manipulator movements
jog(mkv3(v3, 0, 0, -106)); // position gripper to grasp device
zero_force();               // init force sensor
gripper(0.0, 10, 1000);     // close gripper
jog(mkv3(v3, 0, 0, 127));  // move device clear of nest

/**************** check if object was retrieved ***********/
read_user_force(force_vector);
force_z = abs(100*force_vector[2]);// extract z force and scale results
  if (force_z < 5)
   {
    printf("\n Device was not retrieved. Press any key \n \n");
    getch();
    return(0);
   }
printf("\n Device was retrieved \n \n");


jmove(contact_config);      // position manipulator for next task
set_seg_time(0);            // restore default movement speed
```

```
return(1);

} // end function grasp_device


/**********************************************************************
*
*
* Name:      acquire_target
* Arguments: bearing - bearing of closest object to manipulator
* Returns:   1 if object is found
*          0 if no object is found
*
*   acquire_target checks if the environment is stable enough to attempt
* to place the device. At the same time the check is being made, the
device
* is held at a safe distance from the target.  If the environment is
* stable( 20 reading w/ +/- 1 cm readings)for n sonar readings the
* function returns a 1.  If after n readings the environment is not
* stable it returns a 0.
*
* R.A. Raphael   Mar 1998
**********************************************************************/

int acquire_target(int bearing)
{
vect  f_new, v3;
vect6 home_jv;
vect6 search_config;
vect6 v;
frame f;
char c;
int Row, Col, I, i, n;
int BoardNum = 0;
int ULStat = 0;
int PortNum, Direction;
int PowerVal, BitValue;
int accuracy_count = 0;
int delay=100;
int tool_offset = 20; // allows for stand-off distance to object(20 cm)
vect6 joint_vector, grasp_config;
frame tool_frame;
frame desired_frame;
frame grasp_frame;
unsigned DataValue, DataValue_temp;
float  range,range_in, error, RevLevel = (float)CURRENTREVNUM;
float  sum_value;
float  x_posit, y_posit,z_posit; //location of tool frame
float  Tool_Location, Tool_Location_x, Tool_Location_y; // range of tool
from base frame origin
float  desired_frame_vector;

/* Declare UL Revision Level. This is required for CYDIO I/O board. */
ULStat = cbDeclareRevision(&RevLevel);
```

```
// set flags
//acquire_target  = False;

clrscr();
MoveCursor (1, 10);

printf ("The bearing to target is %d \n \n",bearing);
GetCursor (&Col, &Row); // memorize cursor locaion                    //38
mkv6(search_config, bearing*DTOR, 90.0*DTOR, -132.0*DTOR, 183.0*DTOR,
45.0*DTOR, 100.0*DTOR);

set_seg_time(4);          // slow manipulator
jmove(search_config);     // move manipulator limbs out of the way
set_seg_time(0);          // restore fastest segment time
where(joint_vector, &tool_frame); // remember current tool frame matrix

/* Initiate error handling
  Parameters:
    PRINTALL :all warnings and errors encountered will be printed
    STOPALL  :if any error is encountered, the program will stop */
ULStat = cbErrHandling (PRINTALL, STOPALL);

/* configure FIRSTPORTA  for digital input
  Parameters:
    BoardNum    :the number used by CB.CFG to describe this board.
    PortNum     :the input port
    Direction   :sets the port for input or output */

PortNum = FIRSTPORTA;
Direction = DIGITALIN;
ULStat = cbDConfigPort (BoardNum, PortNum, Direction);
ULStat = cbDIn(BoardNum, PortNum, &DataValue);
DataValue = ~DataValue & 0377;

z_posit = 300;   // z location of the manipulator after error is reduced
where(joint_vector, &desired_frame); // memorize current frame for
rotation matrix
accuracy_count = 0;

n = 50;
while ( --n > 1)  // n attempts will be made to stabilize platform to +/-
2cm
 {
  /* Read the 7 bits digital input and display
    Parameters:
      BoardNum    :the number used by CB.CFG to describe this board
      PortNum     :the input port
      DataValue   :the value read from the port   */

  ULStat = cbDIn(BoardNum, PortNum, &DataValue);
  DataValue = ~DataValue & 0377;

  if(DataValue < 91)   // if reading is too low read again
   {
    usec_timer_init(); // init timer
    usec_timer(delay); // invoke timer
    ULStat = cbDIn(BoardNum, PortNum, &DataValue); // call range
information
    DataValue = ~DataValue & 0377;
    if (DataValue< 91)          // if still low set lowest value
      range_in = 91* 0.472;      // convert ranging information
    else                        // if not low convert read value
      range_in = DataValue*0.472; // convert ranging information cm
   }

  else                           // if not low set convert read value
    range_in = DataValue*0.472;  //convert ranging information to cm

  /******************** compute range to target **********************/
  range = range_in;
  where(joint_vector, &tool_frame);
  Tool_Location_x =  tool_frame.p[0];
  Tool_Location_y =  tool_frame.p[1];
  Tool_Location = 0.1 * sqrt( Tool_Location_x *Tool_Location_x +
Tool_Location_y * Tool_Location_y);

  /******************** create desired position vector *****************/
  error = range - Tool_Location;
  desired_frame_vector = 10.0 * (error + Tool_Location - tool_offset);
  desired_frame.p[0] =  desired_frame_vector * cos(bearing*DTOR);
  desired_frame.p[1] =  desired_frame_vector * sin(bearing*DTOR);
  desired_frame.p[2] =  z_posit;

  /* display the value collected from the port */
  MoveCursor (Col, Row);
  printf ("range to target:                   %2.1f cm\n", range);
  printf ("Tool frame location:              %4.1f cm\n", Tool_Location);
  printf ("Error:                            %4.1f cm\n\n",(error -
tool_offset));
  printf ("Range measurements to termination: %d\n",n);

  cmove(&desired_frame);

  if ( (int)Tool_Location > 20) // lower tool frame after tool frame is
at safe dist
    z_posit = 0.0;

  /**************** determine if environment is stable
******************/
  if ( (abs((int)error)-20) < 2 )
   {
    accuracy_count = accuracy_count + 1;
    printf("a count %d",accuracy_count);
   }
  else
   {
    accuracy_count = 0;
   }
  if (accuracy_count > 20)
   {
    printf("environment is stable \n \n");
    return(1);
   }
```

```
 } // end while
return(0);

} // end function acquire_target

/***********************************************************************
*
*
* Name:      make_contact
* Arguments: bearing - bearing of closest object to manipulator
* Returns:   1 if device contacts target
*          0 if device does not make contact.
*
*   make_contact assumes the manipulator has been placed in a
* configuration for approaching the target. The tool frame is advanced
* toward the target until contact is made.
*
* R.A. Raphael  Mar 1998
***********************************************************************
/

int make_contact(int bearing)
{
vect  f_new, v3;
vect6 home_jv;
vect6 v;
frame f;
char c;
float get_data[3];
int Row, Col, I, i;
int BoardNum = 0;
int ULStat = 0;
int PortNum, Direction;
int PowerVal, BitValue;
int n, accuracy_count = 0;
int delay=100;
int tool_offset = 20; // allows for object being placed
vect6 joint_vector, grasp_config;
vect6 force_vector;
frame tool_frame;
frame desired_frame;
frame grasp_frame;
unsigned DataValue, DataValue_temp;
float  range,range_in, error, RevLevel = (float)CURRENTREVNUM;
float  sum_value;
float  x_posit, y_posit,z_posit; //location of tool frame
float  Tool_Location, Tool_Location_x, Tool_Location_y; // range of tool
from base frame origin
float  desired_frame_vector;
float  force_z;

/* Declare UL Revision Level. This is required for CYDIO I/O board. */
ULStat = cbDeclareRevision(&RevLevel);

/* Initiate error handling
ULStat = cbErrHandling (PRINTALL, STOPALL);
```

```
/* set up the display screen */
ClearScreen();
MoveCursor (1, 10);
printf("The device is being deployed \n\n");
GetCursor (&Col, &Row);

/* configure FIRSTPORTA  for digital input
  Parameters:
    BoardNum    :the number used by CB.CFG to describe this board.
    PortNum     :the input port
    Direction   :sets the port for input or output */
PortNum = FIRSTPORTA;
Direction = DIGITALIN;
ULStat = cbDConfigPort (BoardNum, PortNum, Direction);
ULStat = cbDIn(BoardNum, PortNum, &DataValue);
DataValue = ~DataValue & 0377;

where(joint_vector, &desired_frame);
accuracy_count = 0;
zero_force();
n = 60;
while (n-- > 1)
 {
  ULStat = cbDIn(BoardNum, PortNum, &DataValue);
  DataValue = ~DataValue & 0377;

  if(DataValue < 91)
   {
    usec_timer_init();
    usec_timer(delay);
    ULStat = cbDIn(BoardNum, PortNum, &DataValue);
    DataValue = ~DataValue & 0377;
    if (DataValue< 91)
      range_in = 91* 0.472;
    else
      range_in = DataValue*0.472;
   }
  else
    range_in = DataValue*0.472;

  /****************** compute range to target *******************/
  range = range_in;
  where(joint_vector, &tool_frame);
  Tool_Location_x =  tool_frame.p[0];
  Tool_Location_y =  tool_frame.p[1];
  Tool_Location = 0.1 * sqrt( Tool_Location_x *Tool_Location_x +
Tool_Location_y * Tool_Location_y);

  /****************** compute new position vector ****************/
  error = range - Tool_Location;
  desired_frame_vector = 10.0 * (error + Tool_Location - tool_offset);
  desired_frame.p[0] =  desired_frame_vector * cos(bearing*DTOR);
  desired_frame.p[1] =  desired_frame_vector * sin(bearing*DTOR);
  desired_frame.p[2] =  z_posit;
  /* display the value collected from the port */
  MoveCursor (Col, Row);
  printf ("range to target:                      %2.1f cm\n", range);
  printf ("Tool frame location:                 %4.1f cm\n", Tool_Location);
  printf ("Error:                               %4.1f cm\n",(error -
tool_offset));
  printf ("Tool offset:                          %2d\n\n", tool_offset);
  printf ("Range measurements to termination: %2d\n",n);

  cmove(&desired_frame); // move manipulator to desired position

  if ( (abs((int)error)-tool_offset) < 1 )
    accuracy_count = accuracy_count + 1;
  else
    accuracy_count = 0;

  if (accuracy_count > 3)
    tool_offset = (tool_offset-2);

  read_user_force(force_vector);
  force_z = abs(100*force_vector[2]);// extract z force

  if (force_z > 20)
   {
    printf("surface detected \n \n");
    return(1);
   }

 } // end while

return(0); // device was not placed

} // end function make_contact


/**********************************************************************
*
*
* Name:       apply_device
* Arguments: None
* Returns:    None
*
*   apply_device applies a force to the tool frame in the z direction
* for the period of time specified by the variable push_time and then,
* moves the tool frame away from the surface to which the force was
* applied.
*
* R.A. Raphael   Mar 1998
***********************************************************************
/

void apply_device(void)
{
vect  v3;
vect6 v6;
float push_time = 2.5;
```

```
set_stiffness(mkv6(v6, 0.1, 0.1, 0.0, 0.1, 0.1, 0.1));
set_bias_force(mkv6(v6, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0));
push_with_bias(push_time);
gripper(80.0, 10, 0);
hjog(mkv3(v3, 0, 0, -125));

}

/***********************************************************************
*
*
* Name:     return_device
* Arguments: none
* Returns:   1 if device is retrieved
*          0 if object is not retrieved
*
*   return_device returns a known device to a known locaion and positions
* the manipulator for the next task.
*
* R.A. Raphael  Mar 1998
***********************************************************************
/

int return_device(void)
{
vect v3;
vect6 force_vector;
float force_z;
vect6 grasp_vector, contact_config;

clrscr();
MoveCursor (1, 10);
mkv6(grasp_vector, 15.0*DTOR, 48.6*DTOR, -174.3*DTOR, 182.2*DTOR,
59.0*DTOR, 20.0*DTOR);
mkv6(contact_config, 0.0*DTOR, 90.0*DTOR, -132.0*DTOR, 183.0*DTOR,
48.0*DTOR, 100.0*DTOR);

jmove(grasp_vector);        // Maneuver manipulator over device to be
placed
set_seg_time(4);            // slow manipulator movements
jog(mkv3(v3, 0, 0, -80)); // position gripper to grasp device
zero_force();               // init force sensor
set_seg_time(1);

force_z = 0;
/****************** place device **********************/
while ( force_z < 10)
 {
  jog(mkv3(v3, 0, 0, -3));  // move device clear of nest

  /*************** check if object is placed ***********/
  read_user_force(force_vector);
  force_z = abs(100*force_vector[2]);// extract z force
 }
gripper(80.0, 10, 1000);   // open gripper
printf("\n Device is replaced \n \n");
jog(mkv3(v3, 0, 0, 100));
```

```
jmove(contact_config);      // position manipulator for next task
set_seg_time(0);            // restore default movement speed

return(1);

} // end function return_device
```

LIST OF REFERENCES

1. Integrated Motions, Inc., *Zebra-ZERO User's Manual, Version 3.0*, Integrated Motions, Inc., Berkeley, CA, 1994.

2. Hewlett Packard, General Purpose Motion Control ICs Technical Data Sheet, HCTL-1100 Series, Hewlett Packard.

3. John J. Craig, *Introduction to Robotics Mechanics and Control*, Second Edition, Addison-Wesley, Reading, MA., 1989.

4. Polaroid Corporation, *Ultrasonic Ranging System*, Polaroid and Polapulse®, Cambridge, MA, 1992.

5. CyberResearch, Inc., *Digital I/O Boards CYDIO 24, CYDIO 24H, CYDIO 24C User's Manual, Revision 5.0*, CyberResearch, Branford, CT, 1997.

6. CyberResearch, Inc., *CyDAS Universal Driver Library User's Manual, Revision 3.3*, CyberResearch, Branford, CT, 1997.

7. Li, Yangmin, "Hybrid Control Approach to the Peg-in-Hole Problem," *IEEE Robotics and Automation Magazine*, pp. 52-60, June 1997.

8. Fu, K.S., Gonzalez, R.C., and Lee C.S.G., *Robotics: Control, Sensing, Vision, and Intelligence*, McGraw-Hill Book Company, New York, NY, 1987.

9. McCarragher, J., Hovland, G.,Sikka, P, Aigner, P. Austin, D., "Hybrid Dynamic Modeling and Control of Constrained Manipulation Systems," *IEEE Robotics & Automation Magazine*, pp. 27-44, June 1997.

10. L. Tsai and A. Morgan, "Solving the Kinematics of the Most General Six- and Five-degree-of-freedom Manipulators by Continuation Methods," Paper 84-DET-20, ASME Mechanisms Conference, Boston, MA, October, 1984.

11. D. Pieper and B. Roth, "The Kinematics of Manipulators Under Computer Control," *Proceedings of the Second International Congress on Theory of Machines and Mechanisms*, Vol. 2 Zakopane, Poland, 1969, pp. 159-169.

12. B.W. Kernighan, D.M. Ritchie, *The C Programming Language*, Bell Telephone Laboratories, Incorporated, Englewood Cliffs, NJ, 1978.

13. Borland, *Turbo C++ User's Guide*, Borland International, Scotts Valley, CA, 1992.

14. Nomadic Technologies, *The Sensus 500 User's Manual*, Nomadic Technologies, Mountain View, CA, 1997.

15. D. G., "Fine-Motion Planning for Robotic Assembly in Local Contact Spaced," PhD Dissertation, Computer Science Department, U.Mass., Amherst, MA 01003.

16. Dankin, G., Liu, Y., Popplestone, R.J. (1994),A "Multilevel Assembly Planning System," *Proceedings of the 1994 ASME Winter Annual Meeting in Chicago.*

17. T.M. Sobh, B. Benhabib, "Discrete Event and Hybrid Systems in Robotics and Automation: An Overview," *IEEE Robotics and Automation Magazine*, pp. 16-18, June 1997.

18. Pepyne, D.L. and Cassandras, C., "Optimal Dispatching Control for Elevator Systems During Peak Traffic," *35th IEEE Conference on Decision and Control*, Kobe, Japan, December 1996.

19. Back, A., Guckenheimer, J., and Myers, M., A Dynamical Simulation Facility for Hybrid Systems, *Hybrid Systems*, LNCS 736, R. Grossman, A. Nerode, A.P. Ravn, And H. Rischel eds., pp. 255-267, Springer Verlag, 1993.

20. Brockett, R., Hybrid Models for Motion Control Systems, *Essays on Control: perspectives in the Theory and its Applications*, H.L. Trentelman and J.C. Willems, eds., Birkhauser Publishers, Boston, MA, 1993.

21. C.S. Bonaventura, and K.W. Lilly, "A Constrained Motion Algorithm for the Shuttle Remote Manipulator System," *IEEE Control Systems*, pp. 6, October 1995.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center 8725 John J. Kingman Rd., STE 0944 Ft. Belvoir, VA 22060-6218	2
2.	Dudley Knox Library Naval Postgraduate School 411 Dyer Rd. Monterey, CA 93943-5101	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	1
4.	Professor Xiaoping Yun, Code EC/YX Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	2
5.	Professor John Ciezki, Code EC/CY Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	1
6.	LT Roy A. Raphael 453 N. Hale Street Pottstown, PA 19464	2